Lecture (1)

Programming Paradigms

- Imperative programming:
  - performs step by step task by changing state
  - focus on how to achieve the goal
  - Ex:
    - Object oriented programing
    - Parallel processing approach
    - **Procedural programing paradigm** -->both procedures and functions can
      1. modify state
      2. produce side effects
- Declarative programming:
  - expresses logic of computation without talking about its control flow
  - EX:
    - Logic programing
    - Database processing approach
    - **Functional programing** --> functions are
      1. pure
      2. avoiding side effects (Avoid mutable state; focus on immutability)

Introduction

- Logic of Computation:
  - Program verification and proving properties
  - Involves
    - using logical methods --> to ensure that a program behaves as intended
    - proving that the program meets its specifications and that it is free from certain types of errors (runtime errors, security vulnerabilities)
  - Hoare Logic
    - is a formal system/method
    - used in computer science to reason about the correctness of computer programs.

- Computational Logic:
  - direct use of logic as a programming tool
  - involves
    - using logical expressions and constructs directly within programming to --> guide computation and reasoning.

- Logic:
  - means of formalizing the human thought process.
  - how logic serves as a structured framework --> for understanding and representing the way we think.
  - Formalization + Inference.
  - Uses of Logic:
    - Reasoning about the correctness of programs
    - representing problems and solving them

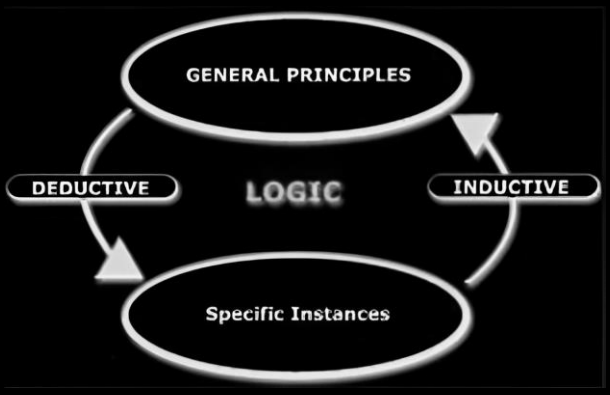
Human Logic

- Logic
  - the science of reasoning
  - deals with the principles of valid inference and correct reasoning
  - a **systematic study** of how we think, argue [Knowledge representation], and draw [Inference] conclusions
- In logical programming --> Reasoning
  - the process of drawing conclusions from facts, rules, and logical statements
  - using logical inference
    - deduce new information.
    - make decisions based on existing knowledge.
- Human brains are information processors.

Reasoning Types

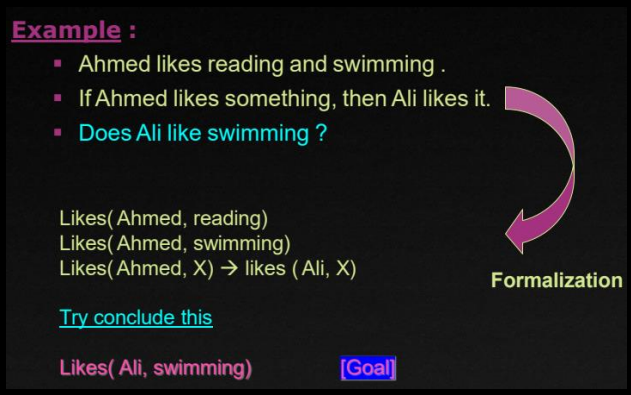
- Deductive Reasoning:
  - from general to particular
  - general rule --> specific conclusion (Always true)
  - the only type that guarantees its conclusion in all cases
  - EX: All men are mortal -> John is a man ->Therefore, John is mortal.
- Inductive Reasoning:
  - from particular to general
  - specific observation --> general conclusion (may be true)
  - EX:
    - I have seen 1000 black cars in my city.
      I have never seen a car that is not black.
      Therefore, every car in my city is black.

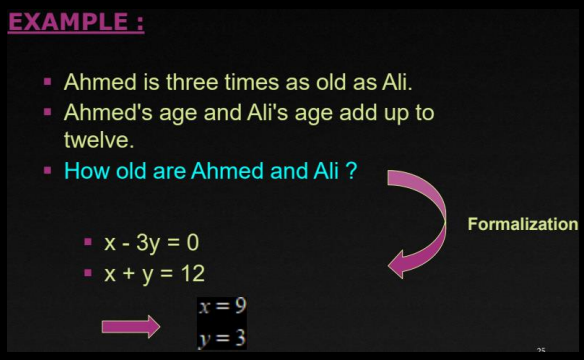
→ Abductive Reasoning:
  → from effects to causes
  → incomplete observation --> best prediction (maybe true)
  → EX:
    → If there is no fuel, the car will not start.
      The car does not start -> Therefore, there is no fuel
→ Reasoning by analogy:
  → based on similarity of two situations.
  → EX:
    → The water flow in a pipe is proportional to its diameter.
      Wires are like pipes -> Therefore, the current in a wire is proportional to its diameter.



Formal Logic

→ a formal version of human logic
→ eliminates natural Languages difficulties using formal language for encoding information.
→ Rules, unambiguous syntax, precise semantic





Applications of Logic

- Artificial Intelligence
- Mathematics --> Automated reasoning programs
  - used to check mathematical proofs
  - used to produce proofs or portions of proofs
- Software analysis --> verify correctness, termination, complexity, etc.
- Law and business
- Hardware Engineering
  - Simulations
  - Configuration and simplification
  - Diagnosis
  - Testing
- Deductive Databases

Logic Programming Paradigm

- Logic Programming Paradigm
  - Python - C
  - focus on
    - numeric types
    - arithmetic operations
    - prioritizing performance for calculations.
- Symbolic Languages:
  - Prolog - Lisp
  - emphasizes
    - logical structures
    - symbolic manipulation
    - reasoning capabilities

Lecture (2)

Propositional logic (**PL**)

- → a declarative sentence that is either true or false
- → A logical model represents propositional sentences & the logical relationships of combining or altering sentences.
- → PL also known as sentential logic
- → propositional statements:
  - → It is raining
  - → 5 is a prime number
  - → 2+2 = 5

  not propositional statements:
  - → Are you hungry?
  - → Shut the door!
  - → X>8
- → Logical operator:
  - → any word or phrase (or – not - if ... then) used to
    - → modify one statement
    - → or join multiple statements
- → Truth-functional **logical operator**:
  - → the truth-values of the statements that used the operator depend on the truth or falsity of the statements from which they are constructed (the result of using this operator is determined by the truth values of the individual statements involved)
  - → EX: (A AND B) whether the whole statement is true or false depends on whether "A" and "B" are true or false.
- → Truth-functional **propositional logic**:
  - → is that branch of propositional logic limited to study Truth-functional logical operator.
- → **Classical** Truth-functional **propositional logic**:
  - → is the branch of Truth-functional propositional logic in which every statement is either true or false, not both.

→ PL Syntax

- → Statement
  - → Simple --> statement letter --> any lower-case letter written with or without a numerical subscript
    EX: p, p2, p_2, p14
  - → Compound --> simple connectives

→ Connective

¬ / ~	Not
& / ^	And
v	or
→	Implication (If then)
↔	Equivalence (If and only if)

→ syntax rules:

1. Any statement letter is a well-formed formula (WFF)
2. α is WFF, then so is (α)
3. α is WFF, then so is ¬α
4. α and β are WFF, then:
   - → α & β
   - → α v β
   - → α → β
   - → α↔β
5. Nothing that cannot be constructed by steps of (1)-(4) is a WFF

EX:

Sentence	PL_Check
p &¬p	wff
pqr	not wff
¬p v	not wff
¬p v ¬p	wff
¬(q v r) ¬q → ¬¬p	not Wff
(p & q) v (p ¬& q)	not Wff

EX: Formalize the following sentences into PL

→ If a person is cool or funny, then he is popular
   → c v f → p

→ Person is popular if and only if he is either cool or funny
   → p ↔ c v f

→ There is no one who is both cool and funny
   → ¬ (c & f)

→ PL Semantic

→ Conjunction: **(α & β)** --> true if both α and β are true and is false otherwise

α	β	(α & β)
T	T	T
T	F	F
F	T	F
F	F	F

→ Disjunction: **(α ∨ β)** --> false if both α and β are false and is true otherwise

α	β	(α ∨ β)
T	T	T
T	F	T
F	T	T
F	F	F

→ Implication: **(α → β)** --> false if α is true and β is false and is true otherwise

α	β	(α⟶β)
T	T	T
T	F	F
F	T	T
F	F	T

→ Equivalence: **(α↔β)** -->true if α and β are either both true or both false and false if they have different truth-values

α	β	(α ↔ β)
T	T	T
T	F	F
F	T	F
F	F	T

→ Negation: **(¬α)** --> true if α is false and false if α is true.

α	¬α
T	F
F	T

→ precedence's rules

1. Negation (¬)
2. Conjunction (∧)
3. Disjunction (∨)
4. Implication (→)
5. Biconditional (↔)

Sentence	Precedence
$p \vee q \wedge r$	$(p \vee (q \wedge r))$
$\neg p \wedge q$	$((\neg p) \wedge q)$
$p \rightarrow q \rightarrow r$	$((p \rightarrow q) \rightarrow r)$
$p \rightarrow q \Leftrightarrow r$	$((p \rightarrow q) \Leftrightarrow r)$

→ Evaluation: the process of determining the truth values of compound sentences given a truth assignment for the truth values of proposition constants

Compute the truth-value for the sentence:
(p ∨ q) ∧ (¬q ∨ r) where **p**, **q** and **r** are **T**, **F**, and **T** respectively.

$$(p \vee q) \wedge (\neg q \vee r)$$
$$(T \vee F) \wedge (\neg F \vee T)$$
$$T \wedge (\neg F \vee T)$$
$$T \wedge (T \vee T)$$
$$T \wedge T$$
$$T$$

→ Satisfaction:
- → the opposite of evaluation
- → figure out which truth assignments satisfy those sentences

EX:

Compute the truth-value for the sentence:
$(p \,\&\, q) \rightarrow \neg r$

p	q	r	p&q	¬ r	(p & q) → ¬r
T	T	T	T	F	F
T	T	F	T	T	T
T	F	T	F	F	T
T	F	F	F	T	T
F	T	T	F	F	T
F	T	F	F	T	T
F	F	T	F	F	T
F	F	F	F	T	T

**If a wff has n distinct statement letters the of possible assignments is 2^n

→ Exclusive OR:
- → Exclusive or in PL is represented by $\neg (\alpha \leftrightarrow \beta)$

α	β	α↔β	¬(α↔β)
T	T	T	F
T	F	F	T
F	T	F	T
F	F	T	F

→ EXAMPLES

(1)

Write the following statement symbolically, and then make a truth table for the statement "If I go to the mall or go to the movies, then I will not go to the gym."

Let

p: I go to the mall
q : I go to the movies
r : I will go to the gym

$(p \vee q) \rightarrow \neg r$

p	q	r	$(p \vee q)$	$\neg r$	$(p \vee q) \rightarrow \neg r$
T	T	T	T	F	F
T	T	F	T	T	T
T	F	T	T	F	F
T	F	F	T	T	T
F	T	T	T	F	F
F	T	F	T	T	T
F	F	T	F	F	T
F	F	F	F	T	T

(2)

Use truth table method to proof the following argument:

p v q
¬p
---
q

p	q	pvq	¬p	(pvq)&¬p	((pvq)&¬p)→q
T	T	T	F	F	T
T	F	T	F	F	T
F	T	T	T	T	T
F	F	F	T	F	T

→ Truth table method drawbacks:
- → Computational complexity
- → Instead, we use
  - → Symbolic Manipulation:
    - → **Resolution:**
      - → A rule of inference
      - → help prove the validity of statements without constructing a complete truth table
    - → **Simplification:**
      - → Reducing complex expressions using identities
      - → like De Morgan's laws

- → Logic Programming:
  - → declarative programming based on logic, reasoning about facts and rules
  - → Approaches like Prolog

Logical Properties and Relationships

- → Tautology (**Valid**)
  - → if and only if it is true for all possible assignments to the statement letters making it up
  - → Ex:

EX: $p \vee \neg p$

p		¬p	p∨¬p
T		F	T
F		T	T

- → Self-contradiction (**unsatisfiable**)
  - → if and only if it is false for all possible truth-value assignments
  - → EX:

EX: $p \& \neg p$

p		¬p	p&¬p
T		F	F
F		T	F

- → Contingent
  - → neither self-contradictory nor tautological.
  - → true for some assignments to its statement letters and false for others.
  - → EX:

EX_1 : $p \& q$
EX_2 : $(p \& q) \rightarrow \neg r$

- → Consistent
  - → if there is at least one possible assignment to the statement letters making them up that makes both wffs true
  - → EX:

EX: $p \vee q$ and $\neg (p \leftrightarrow \neg q)$

p	q		p∨q		¬q	p↔¬q	¬(p ↔ ¬q)
T	T		T		F	F	T
T	F		T		T	T	F
F	T		T		F	T	F
F	F		F		T	F	T

→ Inconsistent
  → if there is no assignment to the statement letters making them up that makes them both true
  → impossible for both formulas to be true at the same time.
  → EX:

EX: $(p \rightarrow q) \& p$ and $\neg(q \vee \neg p)$

p	q	$p \rightarrow q$	$(p \rightarrow q) \& p$	$\neg p$	$q \vee \neg p$	$\neg(q \vee \neg p)$
T	T	T	T	F	T	F
T	F	F	F	F	F	T
F	T	T	F	T	T	F
F	F	T	F	T	T	F

→ Logically equivalent
  → if all possible assignments to the statement letters making them up make them identical
  → EX:

EX: $p \rightarrow q$ and $\neg p \vee q$

p	q	$p \rightarrow q$	$\neg p$	$(\neg p \vee q)$
T	T	T	F	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

PL' (simple version of PL)

→ define all operators of PL using the signs:

→ ¬ and →

AND &	OR ∨	EQUIVALENT ↔
¬ (α → ¬β)	¬α → β	¬ ((α → β) → ¬ (β → α))

→ ¬ and ∨

AND &	Implication →	EQUIVALENT ↔
¬ (¬α ∨ ¬β)	¬α ∨ β	¬ (¬ (¬α ∨ β) ∨ ¬ (¬β ∨ α))

→ ¬ and &

OR ∨	Implication →	EQUIVALENT ↔
¬ (¬α & ¬β)	¬ (α & ¬β)	¬ (α & ¬β) & ¬ (β & ¬α)

PL''

-	'|' NAND (Sheffer stroke)	'↓' NOR (Sheffer dagger)
¬α	α | α	α ↓ α
α & β	(α | β) | (α | β)	(α ↓ α) ↓ (β ↓ β)
α ∨ β	(α | α) | (β | β)	(α ↓ β) ↓ (α ↓ β)
α → β	α | (β | β)	((α ↓ α) ↓ β) ↓ ((α ↓ α) ↓ β)
α ↔ β	((α | α) | (β | β)) | (α | β)	-

Comparison

PL	PL'	PL''
Conforms better with our ordinary reasoning and thinking habits	Simplifies the logical language which makes the work of the deductive systems easier.	
¬ - & - ∨ - → - ↔	¬ - → ¬ - ∨ ¬ - &	| ↓

Lecture (3)

Natural Deduction

→ An attempt to reduce the reasoning to a series of steps
→ each of which is intuitively justified by:
  → the premises of the argument
  → or previous steps

→ consists of a list of intuitively valid rules of inference for
  → the construction of derivations
  → step-by-step deductions

→ EX:

Consider the following argument stated in natural language:

- Today is Tuesday or Wednesday
- The doctor office is open today
- If it is Wednesday, then the doctor office is closed.
- Therefore, Today is Tuesday

$$\begin{array}{l} t \vee w \\ \neg d \\ \underline{w \rightarrow d} \\ t \end{array}$$

Rules Of Inference

→ Modus ponens (MP)
  → the rule of "→-elimination"
  → affirm the consequent of a conditional statement when its antecedent is true

$$\begin{array}{l} \alpha \rightarrow \beta \\ \alpha \\ \hline \beta \end{array}$$

→ Modus tollens (MT)
  → the rule of "→-elimination"
  → deny the antecedent of a conditional statement when its consequent is false.

$$\begin{array}{l} \alpha \rightarrow \beta \\ \neg \beta \\ \hline \neg \alpha \end{array}$$

antecedent
Consequent

→ Disjunctive syllogism (DS)

→ the rule of "v-elimination "

$$\frac{\alpha \vee \beta \qquad \neg\alpha}{\beta}$$

→ Addition (Add.)

→ the rule of "v-Introduction "

$$\frac{\alpha}{\alpha \vee \beta}$$

→ Simplification (Simp.)

→ the rule of "&-elimination "

$$\frac{\alpha \,\&\, \beta}{\alpha} \qquad \frac{\alpha \,\&\, \beta}{\beta}$$

→ Conjunction (Conj.)

→ the rule of "&- introduction "

$$\frac{\alpha \qquad \beta}{\alpha \,\&\, \beta}$$

→ Hypothetical syllogism (HS)

→ the rule of "chain deduction "

$$\frac{\alpha \rightarrow \beta \qquad \beta \rightarrow \gamma}{\alpha \rightarrow \gamma}$$

Example:

- If it rains (P), then the streets will be wet (Q).
- If the streets are wet (Q), then there will be puddles برك (R).
- Therefore, if it rains (P), then there will be puddles (R).

→ Absorption (Abs.)

$$\frac{\alpha \rightarrow \beta}{\alpha \rightarrow (\alpha \,\&\, \beta)}$$

Example:

Premise: If it is sunny (P), then the ground is dry (Q).

Absorption Application: From P→Q, you can infer P→(P∧Q).

This means:

- If it is sunny, then it is sunny and the ground is dry. This conjunction reinforces the idea that being sunny directly correlates with the ground being dry.

→ Constructive dilemma (CD)

→ enables reasoning from

→ multiple conditional statements

→ and a disjunction

→ disjunction introduction:

→ (v-introduction)

→ allows you to add a disjunction based on one true proposition

→ constructive dilemma

→ combines multiple conditional statements → to derive a conclusion involving disjunction.

$$\frac{\alpha \rightarrow \gamma \qquad \beta \rightarrow \delta \qquad \alpha \vee \beta}{\gamma \vee \delta}$$

Rules Of Replacement

- → Double negation (DN):
  - → ¬ ¬ α → α
- → Commutativity (Com.):
  - → α & β → β & α
  - → α ∨ β → β ∨ α
- → Tautology (Taut.):
  - → α → α & α
  - → α → α ∨ α
- → Material Implication (Impl.):
  - → α → β → ¬ α ∨ β
- → Distribution (Dist.):
  - → α & (β ∨ γ) → (α & β) ∨ (α & γ)
  - → α ∨ (β & γ) → (α ∨ β) & (α ∨ γ)
- → Transposition (Trans.):
  - → α → β → ¬β → ¬ α
- → Associativity (Assoc.):
  - → (α & β) & γ → α & (β & γ)
  - → (α ∨ β) ∨ γ → α ∨ (β ∨ γ)
- → DeMorgan's Laws (DM):
  - → ¬ (α & β) → ¬ α ∨ ¬β
  - → ¬ (α ∨ β) → ¬ α & ¬β
- → Material Equivalence (Equiv.):
  - → α ↔ β → (α → β) & (β → α)

Comparison

Rules of Inference	Rules of Replacement
Inferring something new	Stating the same thing using a different combination of symbols.
→ Apply when the main operators match the patterns given → and only apply to the entire statement.	→ Can be applied to portions of statements
Unidirectional	**Bidirectional**

Reasoning Procedures

- Direct Deductions (DD)
  - consists of an ordered sequence of wffs each one is either
    - a premise
    - derived from previous members by
      - one of the inference rules
      - the replacement rules
    - The conclusion is the final step of the sequence
  - EX:

(1)

1) t ∨ w	Premise
2) ¬d	Premise
3) w → d	Premise
4) ¬w	2,3 MT
5) t	1,4 DS #

t v w
¬ d
w → d
---
t

(2)

Solution 1

1) c ∨ d	Premise
2) c → o	Premise
3) d → m	Premise
4) ¬o	Premise
5) ¬c	2,4 MT
6) d	1,5 DS
7) m	3,6 MP #

c v d
c → o
d → m
¬o
---
m

Solution 2

There is no unique derivation for a given conclusion from a given set of premises.

1) c ∨d	Premise
2) c → o	Premise
3) d → m	Premise
4) ¬o	Premise
5) (c → o) & (d → m)	2,3 Conj.
6) o ∨ m	1,5 CD
7) m	4,6 DS #

- Conditional proof (CP)
  - derivation technique used to establish a conditional wff using '→' operator
  - by constructing a sub derivation within a derivation
  - Assume the **antecedent** as a hypothesis
  - If the consequent is derived --> end the sub-derivation and conclude the conditional statement (P→QP \to QP→Q) in the main derivation
  - EX:

p → (q v r)
p → ¬s
s ↔ q
---
p → r

Establish the validity of this argument using CP.

1) p → (q ∨ r)	Premise
2) p → ¬s	Premise
3) s ↔ q	Premise
4) p	Assumption , antecedent
5) q ∨ r	1,4 MP
6) ¬s	2,4 MP
7) (s → q) & (q → s)	3 Equiv.
8) q → s	7 Simp.
9) ¬q	6,8 MT
10) r	5,9 DS , consequent
11) p → r	4-10 CP#

→ Indirect proof (IP)

- → demonstrate that a certain wff is false on the basis of the premises.
- → assuming the opposite of that which we're trying to prove.
- → If we can demonstrate an obvious contradiction, we can conclude that the assumed statement must be false
- → because anything that leads to a contradiction must be false.

EX:

For example, consider the following argument:

$$p \rightarrow q$$
$$\underline{p \rightarrow (q \rightarrow \neg p)}$$
$$\neg p$$

1) $p \rightarrow q$	Premise
2) $p \rightarrow (q \rightarrow \neg p)$	Premise
3) p	Assumption (negation of the conclusion)
4) q	1,3 MP
5) $q \rightarrow \neg p$	2,3 MP
6) $\neg p$	4,5 MP
7) $p \ \& \ \neg p$	3,6 Conj. ,create contradiction
8) $\neg p$	3-7 IP# ,the opposite assumption is true

Lecture (4)

PL drawbacks

- not powerful enough to represent
  - all types of assertions --> used in computer science and mathematics --> X > 1
  - certain types of relationship between propositions --> equivalence -->"Not all integers are even", "some integers are not even"
- To solve the problem:
  - Predicates
  - Quantifiers

Predicate

- is a verb phrase template that describes:
  - property of objects.
  - or a relationship among objects
- A predicate with variables is called an atomic formula. P(X)
- Every atomic formula has a degree (arity)
  - A unary --> arity 1 (P (X))
  - A binary --> arity 2 (R (X, Y))
  - A ternary --> arity 3 (S (X, Y, Z))
- EX:

(1)

The car Tom is driving is blue The sky is blue The cover of book is blue	blue(tom's car) is_blue(sky) b(book's cover)

(2)

Ahmed is the parent of Ali Ahmed gives a book to Ali	parent(ahmed, ali) give(ahmed, book, ali)

(3)

If John is a cool man, then kay is a cool man

$$Cool(john) \rightarrow Cool(kay)$$

john is not a cool man

$$\neg\ Cool(john)$$

neither john nor Kay is a cool man

$$\neg\ Cool(john)\ \&\ \neg Cool(kay)$$

Quantifiers

- → A predicate with variables (called an atomic formula) is not a proposition
- → x >1 --> to be a proposition
  - → Assign a value to the variable
  - → Or quantify the variable
- → universe of discourse --> set of objects of interest
- → universe --> is the domain of the (individual) variables

Universal Quantifier ∀x	Existential Quantifier ∃x
→ For all x – for each x – for every x → means all objects x in the universe → P(X) is true for every object x	→ There exists an x - There is at least one x → means at least one object x in the universe → P(X) is true for at least one object x
equivalent to the **conjunction (&)**: → P(x1) & P(x2) & P(x3) & … P(xn)	equivalent to the **disjunction (V)**: → P(x1) v P(x2) v P(x3) v … $P(x_n)$

- → How to read quantified formulas

P(X) denote: "x is Perfect"

∀x P(X) translated to:

- "For every object x the following holds: x is perfect
- "Everything is perfect"

∃x P(X) translated to:

- "For some object x the following holds: x is perfect"
- " Some thing is perfect"

∀ x¬P(X) translated to:

- " Every things is not perfect"

¬∀x P(X) translated to:

- "Not every thing is perfect"

P(book) translated to:

- "this book is perfect"

→ A bound variable:
  → is a variable that is either
    → assigned a specific value
    → or is quantified by a universal (∀) or existential (∃) quantifier
  → If variable is not bound → it is called free

EX: $\exists x\, q(X,Y)$: Y is free, and X is bound

→ The scope of the quantifier
  → [ ]
  → No [ ] --> The scope is the smallest wff following the quantification
  → EX:

In ∃x P(x, y):

the variable **x** is bound while **y** is free.

In ∀x [∃y P(x, y) v Q(x, y)]:

x and the y in P(x, y) are **bound**, while **y** in Q(x, y) is **free**, because the scope of ∃y is P(x, y) and the scope of ∀x is [∃y P(x, y) v Q(x, y)].

→ How to read quantified formulas

EX:

(1)

L(X,Y) denote: "X likes Y"

∀x ∀y L(X, Y) translated to:
"Every one Like every one"

∀x L(X, ali) translated to:
"Every one Like ali"

∀x ∃y L(X, Y) translated to:
"Every one Like some one"

∃x ∀y L(X, Y) translated to:
"some one Like every one" [at least one person]

19

(2)

Translating English to Predicate Logic WFFs

- **" Ahmed likes every one who Ali likes "**
  ∀x [L(ali,X) → L(ahmed,X)]
- **" Ahmed likes every one who likes him"**
  ∀x [L(X,ahmed) → L(ahmed,X)]
- **" Ahmed likes some one who like him"**
  ∃x [L(ahmed, X) & L(X, ahmed)]

(3)

Translating English to Predicate Logic WFFs:

E(x): x is even , **O(x):** x is odd

- **"Not every integer is even"**
  $\neg \forall x\, E(x)$
- **"Some integers are not even"**
  $\exists x\, \neg E(x)$
- **"Some integers are even and some are odd "**
  $\exists x\, E(x) \wedge \exists x\, O(x)$

- **"If an integer is not even, then it is odd"**

$\forall x\, [\neg E(x) \rightarrow O(x)\,]$

(4)

Translating English to Predicate Logic WFFs:

I(x): x is integer, **E(x):** x is even , **O(x):** x is odd

- **"All integers are even"**
  $\forall x\, [\, I(x) \rightarrow E(x)\,]$
- **"Some integers are odd"**
  $\exists x \quad [I(x) \wedge O(x)]$
- **"Only integers are even" ≡ "if it is even, then it is integer"**
  $\forall x\, [\, E(x) \rightarrow I(x)\,]$

First Order Logic (FOL)

Predicate Logic

→ syntax rules

- → Every atomic formula is wff
- → If α wff --> then so is $[\alpha]$
- → If α wff --> then so is $\neg\alpha$
- → If α and β wff --> then the following are wff:
  - → $\alpha \,\&\, \beta$
  - → $\alpha \vee \beta$
  - → $\alpha \rightarrow \beta$
  - → $\alpha \leftrightarrow \beta$
- → If x is a wff --> then the following are wff:
  - → $\forall x\, \alpha$
  - → $\exists x\, \alpha$
- → Nothing else is a wff

→ Interpretation

→ A (WFF) is generally not a proposition by itself

→ It becomes a proposition only when it is interpreted with specific values or a domain for its variables

→ EX: ∀x P(X) → where P(X) denote: "X is Positive "

→ Evaluation

→ determining the truth values

→ EX:

- Suppose we have two predicates p(x), q(x,y), and the universe of discourse is the set {a,b}
- Suppose the following truth assignment:

p(a): T	p(b): F
q(a,a): T	q(b,b): F
q(a,b): F	q(b,a): T

- The truth assignment for the wff: **∀x [p(X) → q(X,X)]**

[p(a)→q(a,a) & p(b) →q(b,b)]
[(T →T) & (F → F)]
[T & (F → F)]
[T & T]
T

26

- The truth assignment for the wff: **∀x ∃y q(X,Y)** is:

[∃y q(a,Y) & ∃y q(b,Y)]
[[q(a,a) ∨ q(a,b)] & [q(b,a) ∨ q(b,b)]]
[[T∨ F] & [T ∨ F]]
[T & [T ∨ F]]
[T & T]
T

27

→ Satisfaction

→ satisfiable if there exists an interpretation that makes it true

→ A wff is valid if it is true for every interpretation

→ A wff is unsatisfiable if it is false for every interpretation

p(a) ∨ p(b)
∀x [p(X) → q(X)]
∃x q(X)

Truth table size: 2^{m*n^k}

For a language with *n* constants and *m* relation of arity *k*

p(a)	p(b)	q(a)	q(b)	p(a) ∨ p(b)	∀x.(p(x) ⇒ q(x))	∃x.q(x)
1	1	1	1	1		1
1	1	1	0	1		1
1	1	0	1	1		1
1	1	0	0	1		0
1	0	1	1	1		1
1	0	1	0	1		1
1	0	0	1	1		1
1	0	0	0	1		0
0	1	1	1	1		1
0	1	1	0	1		1
0	1	0	1	1		1
0	1	0	0	1		0
0	0	1	1	0		1
0	0	1	0	0		1
0	0	0	1	0		1
0	0	0	0	0		0

→ Equivalence:

→ if and only if W1 ↔ W2 is valid, that is if and only if W1 ↔ W2 is true for all interpretations

→ EX:

→ ∀x P(x) and ¬ ∃x ¬P(x) are equivalent for any predicate name P.

→ ∀x [P(x) ∧ Q(x)] and ∀x P(x) ∧ ∀x Q(x) are equivalent

→ Reasoning in FOL (Axioms of predicate logic)

→ Universal instantiation (**UI**):

$$\frac{\forall x\, P(x)}{P(c)}$$

→ c is one arbitrary element of the universe

→ Universal Generalization (**UG**):

$$\frac{P(c)}{\forall x\, P(x)}$$

→ P(c) holds for every element x of the universe of discourse

→ Existential Instantiation (**EI**):

$$\frac{\exists x\, P(x)}{P(c)}$$

→ c is some element of the universe of discourse

→ It is not arbitrary

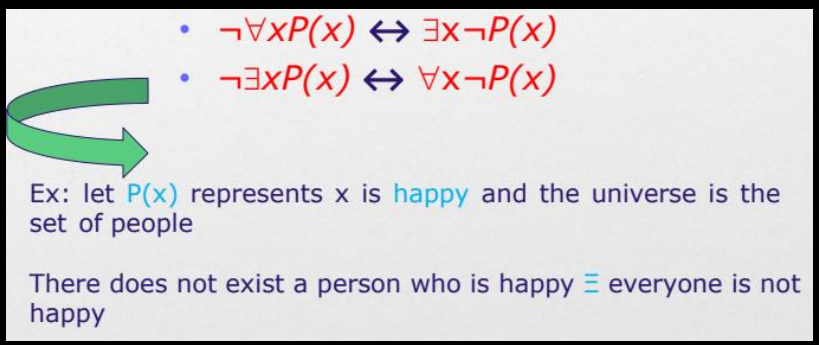
→ but must be one for which P(c) is true

→ Existential Generalization(**EG**):

$$\frac{P(c)}{\exists x\, P(x)}$$

→ C is an element of the universe

→ Negation of quantified statement:



→ Examples:

(1)

consider the following argument:

$$\frac{\forall x[P(X) \,\&\, G(X)]}{\forall x P(X) \,\&\, \forall x\, G(X)}$$

- **Predicates :**
  - P(x): x is Positive
  - G(x): x is greater than zero
  - The universe is the set {1,2,3}

1) ∀x[P(X)&G(X)]	Premise
2) P(1)&G(1)	1 UI
3) P(1)	2 Simpl.
4) G(1)	2 Simpl.
5) P(2)&G(2) 6) P(2) 7) G(2) 8)... 9)...10).. {3}	
11) ∀x P(X) 12) ∀x G(X)	3,6,9 UG 4,7,10 UG
13) ∀xP(X) & ∀x G(X)	5,6 Conj #

(2)

A check is void لاغي if it has not been cashed for 30 days. This check has not been cashed for 30 days. You cannot cash a check which is void. Show that we now have a check which cannot be cashed.

$$C(\text{this\_check})$$
$$\neg T(\text{this\_check})$$
$$\forall x\, [\,[C(X) \,\&\, \neg T(X)] \rightarrow V(X)]$$
$$\forall x\, [\,[C(X) \,\&\, V(X)] \rightarrow \neg S(X)]$$

$$\therefore \exists x\, [\, C(X) \,\&\, \neg S(X)]$$

Predicates :

- C(x): x is a check.
- T(x): x has been cashed within 30 days.
- V(x): x is void.
- S(x): x can be cashed.

1	C(this_check)	Premise
2	¬T(this_check)	Premise
3	∀x[[C(X)& ¬T(X)] →V(X)]	Premise
4	∀x[[C(X)&V(X)] →¬S(X)]	Premise
5	C(this_check)&¬T(this_check)	1,2 Conj.
6	[C(this_check)& ¬T(this_check)] →V(this_check)	3 UI
7	V(this_check)	5,6 MP
8	C(this_check)& V(this_check)	1,7 Conj.
9	C(this_check)& V(this_check) →¬S(this_check)	4 UI
10	¬S(this_check)	9,8 MP
11	C(this_check) & ¬S(this_check)	1,10 Conj.
12	∃x[C(X)& ¬S(X)]	11 EG #

Lecture (5)

Propositional resolution:

- Why?
  - Inference rules and replacement rules are complex
  - use only one rule to build a complete inference procedure.
  - used as the inference engine in PROLOG
  - only applies to clausal form
  - build a theorem prover that is sound and complete for all propositional logic.

- Clausal Form
  - Literal
    - statement letter
    - or a negation of statement letter
    - P
      ¬p
  - clause expression
    - a literal
    - or a disjunction of literals
    - P
      p ∨ q
  - clause
    - the set of literals in a clause expression
    - {p}
      {p, q}
  - Contradiction
    - (False)
    - represented by empty clause { } or ∅

→ Converting PL to CNF

→ Resolution rule only applies to disjunctions

→ a formula is in CNF

→ Its only connectives are V, &, ¬

→ ¬ applies only to statement letters

→ Formulas are on product of sum form

1. Implications (I):

$$\varphi 1 \Rightarrow \varphi 2 \quad \neg\varphi 1 \vee \varphi 2$$

$$\varphi 1 \Leftrightarrow \varphi 2 \quad \rightarrow \quad (\neg\varphi 1 \vee \varphi 2) \wedge (\varphi 1 \vee \neg\varphi 2)$$

2. Negations (N):

$$\neg\neg\varphi \rightarrow \varphi$$
$$\neg(\varphi 1 \wedge \varphi 2) \rightarrow \neg\varphi 1 \vee \neg\varphi 2$$
$$\neg(\varphi 1 \vee \varphi 2) \rightarrow \neg\varphi 1 \wedge \neg\varphi 2$$

3. Distribution (D):

$$\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \longrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$$

4. Operators (O):

$$\varphi_1 \vee \ldots \vee \varphi_n \longrightarrow \{\varphi_1, \ldots, \varphi_n\}$$
$$\varphi_1 \wedge \ldots \wedge \varphi_n \longrightarrow \{\varphi_1\} \ldots \{\varphi_n\}$$

→ Examples

(1)

$(g \wedge (r \Longrightarrow f))$

	$g \wedge (r \Longrightarrow f)$
I	$g \wedge (\neg r \vee f)$
N	$g \wedge (\neg r \vee f)$
D	$g \wedge (\neg r \vee f)$
O	$\{g\}$
	$\{\neg r, f\}$

(2)

$\neg(g \wedge (r \Longrightarrow f))$

	$\neg(g \wedge (r \Longrightarrow f))$
I	$\neg(g \wedge (\neg r \vee f))$
N	$\neg g \vee \neg(\neg r \vee f))$
	$\neg g \vee (\neg\neg r \wedge \neg f)$
	$\neg g \vee (r \wedge \neg f)$
D	$(\neg g \vee r) \wedge (\neg g \vee \neg f)$
O	$\{\neg g, r\}$
	$\{\neg g, \neg f\}$

Propositional Resolution

→ propositional resolution

$$\frac{\{p, q\} \quad \{\neg p, r\}}{\{q, r\}}$$

$$\frac{\{\varphi_1, \ldots, \chi, \ldots, \varphi_m\} \quad \{\psi_1, \ldots, \neg\chi, \ldots, \psi_n\}}{\{\varphi_1, \ldots, \varphi_m, \psi_1, \ldots, \psi_n\}}$$

→ Multiple occurrences

$$\frac{\{\neg p, q\} \quad \{p, q\}}{\{q\}}$$

→ Empty clause

$$\frac{\{p\} \quad \{\neg p\}}{\{\}}$$

→ Resolvent

$$\frac{\{p, q\} \quad \{\neg p, \neg q\}}{\{\} \text{ Wrong!}}$$

$$\frac{\{p, q\} \quad \{\neg p, \neg q\}}{\{p, \neg p\} \quad \{q, \neg q\}}$$

→ Resolution Vs. Inference Rule

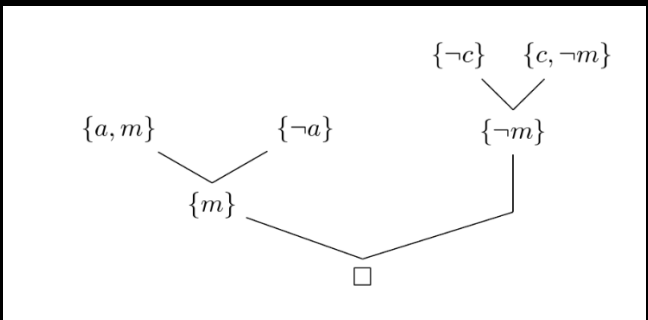
Disjunctive syllogism (DS)	Hypothetical Syllogism (HS)	Modus Tollens (MT)
$\frac{p \vee q \quad \neg q}{p}$ $\quad$ $\frac{\{p, q\} \quad \{\neg q\}}{\{p\}}$	$\frac{p \Rightarrow q \quad q \Rightarrow r}{p \Rightarrow r}$ $\quad$ $\frac{\{\neg p, q\} \quad \{\neg q, r\}}{\{\neg p, r\}}$	$\frac{p \Rightarrow q \quad \neg q}{\neg p}$ $\quad$ $\frac{\{\neg p, q\} \quad \{\neg q\}}{\{\neg p\}}$

→ EX:

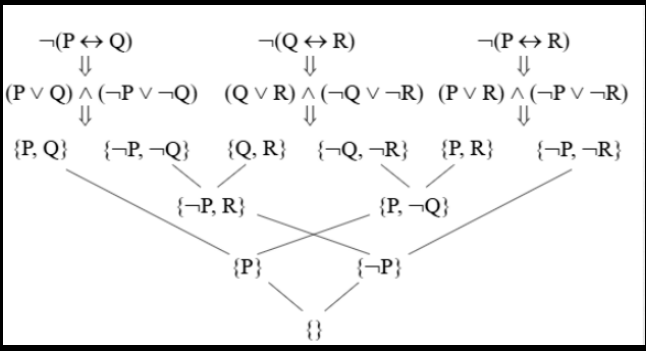
In order to determine whether a set of clauses is unsatisfiable, we look for a resolution proof of the empty clause from the set of premises

1) {p,q}	Premise
2) {¬p,q}	Premise
3) {p, ¬q}	Premise
4) {¬p,¬q}	Premise
5) {q}	1,2
6) {¬q}	3,4
7) { }	5,6

→ Resolution Tree:



→ Resolution Graph



→ Resolution Algorithm

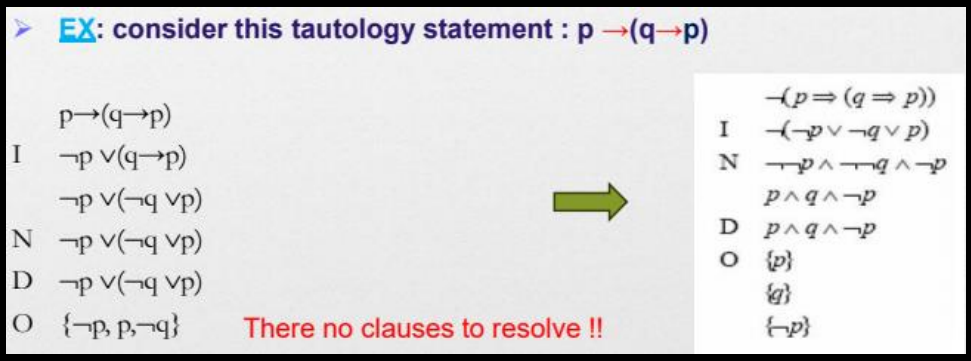
```
function PL-RESOLUTION(KB, α)
    returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
        α, the query, a sentence in propositional logic
clauses ← the set of clauses in the CNF representation of KB ∧
¬α
new ← {}
loop do
    for each pair of clauses Ci, Cj in clauses do
        resolvents ← PL-RESOLVE(Ci, Cj)
        if resolvents contains the empty clause then return true
        new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

→ Validity Checking

→ The resolution process is incomplete for tautologies because it relies on finding contradiction.

→ resolution proof is generally **not** complete

EX:



→ Refutation

→ When resolution is used to prove inconsistency or unsatisfiable

→ Refutation is complete

→ Inconsistent

→ If we derive a contradiction --> the conclusion follows from the premises.

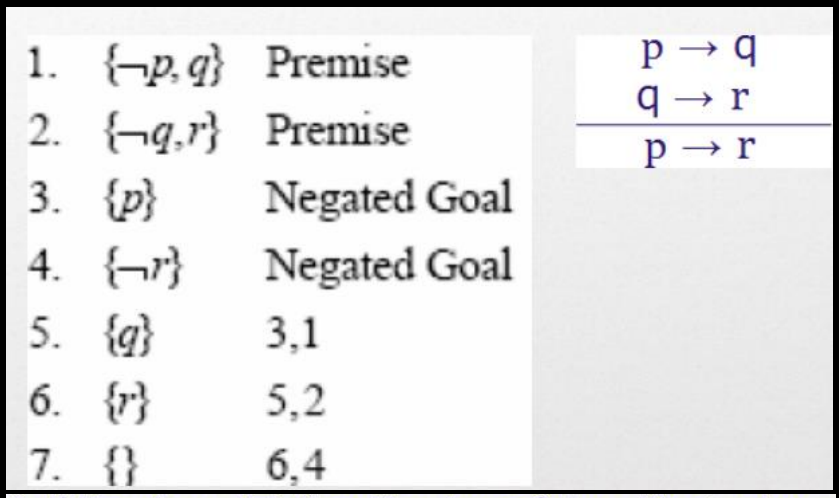
→ All inconsistencies can be reached by refutation.

→ Consistent

→ If **no** contradiction can be derived

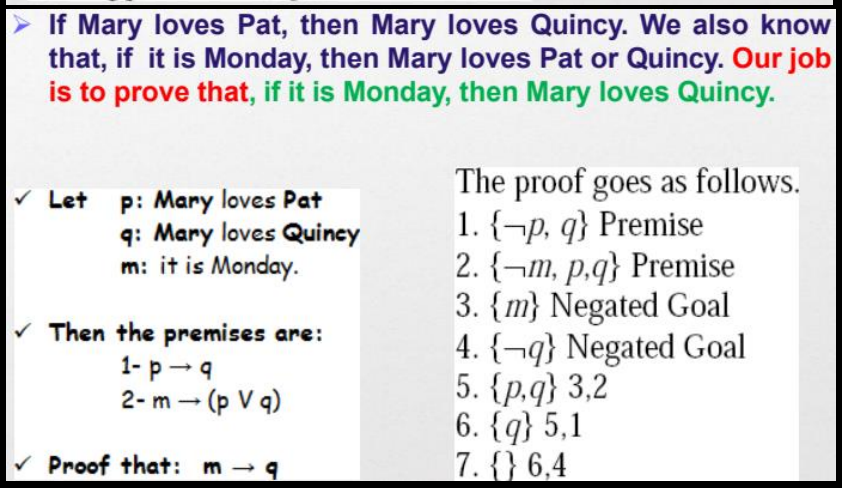
→ If we can't apply any more --> the conclusion cannot be proved from the premises

→ resolution algorithm
  → uses the principle of proof by contradiction
  → negate the goal and then add it to the premises



1.	$\{\neg p, q\}$	Premise
2.	$\{\neg q, r\}$	Premise
3.	$\{p\}$	Negated Goal
4.	$\{\neg r\}$	Negated Goal
5.	$\{q\}$	3,1
6.	$\{r\}$	5,2
7.	$\{\}$	6,4

$$\frac{p \rightarrow q \quad q \rightarrow r}{p \rightarrow r}$$



- If Mary loves Pat, then Mary loves Quincy. We also know that, if it is Monday, then Mary loves Pat or Quincy. Our job is to prove that, if it is Monday, then Mary loves Quincy.

- ✓ Let p: Mary loves Pat
  q: Mary loves Quincy
  m: it is Monday.
- ✓ Then the premises are:
  1- $p \rightarrow q$
  2- $m \rightarrow (p \vee q)$
- ✓ Proof that: $m \rightarrow q$

The proof goes as follows.
1. $\{\neg p, q\}$ Premise
2. $\{\neg m, p, q\}$ Premise
3. $\{m\}$ Negated Goal
4. $\{\neg q\}$ Negated Goal
5. $\{p, q\}$ 3,2
6. $\{q\}$ 5,1
7. $\{\}$ 6,4

→ Resolution Provability

A sentence φ is *provable* from a set of sentences Δ by propositional resolution (written $\Delta \vdash \varphi$) if and only if there is a derivation of the empty clause from the clausal from of $\Delta \cup \{\neg\varphi\}$.

Resolution Strategies

→ When doing resolution automatically
  → decide in which order to resolve the clauses
  → as it affects the time needed to find a contradiction.

→ Unit preference:
  → choose a resolution step involving a unit clause (clause with one literal)
  → preference is given to clauses that have not been used yet.
  → Produces a shorter clause

→ not complete → when none of the premises contain a unit clause

→ EX:

1)	$\{\neg p_1, p_2\}$	Premise
2)	$\{\neg p_2\}$	Premise
3)	$\{p_1, p_3, p_4\}$	Premise
4)	$\{\neg p_3, p_5\}$	Premise
5)	$\{\neg p_6, \neg p_5\}$	Premise
6)	$\{p_6\}$	Premise
7)	$\{\neg p_4\}$	Negated goal
8)	$\{\neg p_1\}$	1,2
9)	$\{\neg p_5\}$	5,6
10)	$\{p_1, p_3\}$	3,7
11)	$\{p_3\}$	8,10
12)	$\{\neg p_3\}$	4,9
13)	$\{\ \}$	11,12

→ Set of support:

→ Choose a resolution involving the negated goal or any clause derived from the negated goal

→ set of support strategy is complete

EX:

Prove P_4 from $P_1 \rightarrow P_2, \neg P_2, \neg P_1 \rightarrow P_3 \vee P_4, P_3 \rightarrow P_5, P_6 \rightarrow \neg P_5$ and P_6, by using the set of support strategy.

Initially the set of support is given by $\neg P_4$, the negation of the conclusion.

One then does resolutions involving $\neg P_4$, then possible resolutions involving the resulting resolvents, and so on.

1.	$\neg P_1 \vee P_2$	Premise
2.	$\neg P_2$	Premise
3.	$P_1 \vee P_3 \vee P_4$	Premise
4.	$\neg P_3 \vee P_5$	Premise
5.	$\neg P_6 \vee \neg P_5$	Premise
6.	P_6	Premise
7.	$\neg P_4$	Negation of conclusion

8.	$P_1 \vee P_3$	7, 3
9.	$P_2 \vee P_3$	1, 8
10.	P_3	2, 9
11.	P_5	4, 10
12.	$\neg P_6$	5, 11
13.	$\emptyset$	6, 12

→ Davis-Putnam Procedure (DPP):

→ treats clauses as sets

→ DPP strategy is complete.

→ given input of nonempty set repeats the following steps until there are no variables left:

1. Choose a variable P appearing in one of the clauses
2. Add all possible resolvents using resolution on P to the set of clauses

3. Discard all clauses with P or ¬ P in them

→ EX:

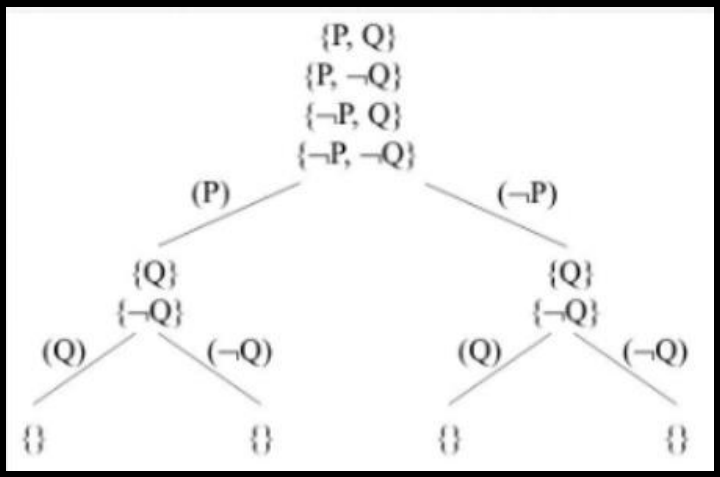
Let us apply the Davis-Putnam procedure to the clauses

$$\{\neg P, Q\}, \{\neg Q, \neg R, S\}, \{P\}, \{R\}, \{\neg S\}$$

- Eliminating P gives $\{Q\}, \{\neg Q, \neg R, S\}, \{R\}, \{\neg S\}$
- Eliminating Q gives $\{\neg R, S\}, \{R\}, \{\neg S\}$.
- Eliminating R gives $\{S\}, \{\neg S\}$.
- Eliminating S gives $\{\}$.

So the output is the empty clause.

TREE



Lecture (6)

Predicate (Relational) Resolution

- Predicate Resolution
  - is a rule of inference used in **relational logic**
  - which deals with relations between objects rather than specific truth values.
  - we can build a reasoning program that is both **sound** and **complete** for all relational logic.
  - Soundness
    - Any conclusion you derive from the premises is true according to the relationships specified in the logic
    - If you derive a conclusion from the premises, it must be true.
  - Completeness
    - If something is logically implied by the premises (i.e., it is true in all models of the system), the relational resolution process can derive that conclusion
    - If a conclusion is true, the system can derive it.

- Not Sound:
  - if the rule leads to incorrect conclusions based on true premises, it is not sound

Example: Affirming the Consequent

- Rule: If P⇒Q and Q is true, then P is true.
- This is a logical fallacy مغالطة منطقية called **affirming the consequent**. The rule seems plausible at first glance but is **not sound** because it can lead to incorrect conclusions.
  - Premise 1: "If it is raining, the ground will be wet" (P⇒Q).
  - Premise 2: "It is not raining" (¬P).
  - Conclusion: "The ground will not be wet" (¬Q).
  - This conclusion is **not guaranteed** because the ground could still be wet for reasons other than rain (e.g., someone watering the plants)
  - F⇒T (False implies True) is true in formal logic because when the premise is false, the implication is considered true regardless of the truth value of the conclusion

الصحّة تعني أن النظام لا يؤدي إلى استنتاجات خاطئة.

4

→ Not Complete:

→ if there are true conclusions that cannot be derived using that rule

→ The rule doesn't cover all logical conclusions that can be made from the premises.

Example: Modus Tollens with Negation of the Conclusion

- Rule: If P⇒Q, and we know ¬Q, then we can infer ¬P (this is Modus Tollens). However, a rule that applies to only some types of negation or that limits what can be negated might not be complete.
  - Premise 1: "If it is raining, then the ground is wet" (P⇒Q).
  - Premise 2: "The ground is not wet" (¬Q).
  - Conclusion: "It is not raining" (¬P).
  - This is a valid use of Modus Tollens, but imagine a scenario where we **cannot negate certain components** directly. This would make the rule **incomplete**, as it would miss some valid inferences.

الكمال يعني أن النظام يمكنه استنتاج كل الحقائق التي يمكن استنتاجها منطقيًا من المقدمات

Unification

→ determining whether two expressions can be made identical by substituting variables with appropriate terms.

→ Substitution

→ a finite mapping of variables to terms

→ Expressed by

{X ← a, Y ← b, Z ← v}

X is replaced by a , Y is replaced by b , Z is replaced by v

→ Domain → The variables being replaced

→ Range → the terms replacing them

EX: {X ← a, Y ← b, Z ← v}

the domain is {X, Y, Z}, and the range is {a, b, v}

→ Substitution is

→ pure

→ iff The replacement terms do not contain the variables being substituted.

→ {X ← a, Y ← b, Z ← v} Pure substitution

→ Impure

→ The replacement terms contain the variables being substituted.

→ {X ← a, Y ← b, Z ← X} Impure substitution

→ The result of applying substitution σ to an expression ϕ
  → is the expression ϕσ
  → obtained by replacing every occurrence of every variable in the domain of the substitution by the term with which it is associated.
  → EX:

q(X,Y) {X ⟵ a, Y ⟵ b} = q(a,b)

q(X,X) {X ⟵ a, Y ⟵ b} = q(a,a)

q(X,W) {X ⟵ a, Y ⟵ b, Z ⟵ v} = q(a,W)

q(Z,v) {X ⟵ a, Y ⟵ b, Z ⟵ v} = q(v,v)

  → If substitution is **pure**
    → application is **idempotent** → applying a substitution a second time has no effect.
    →

q(X,X,Y,W,Z) {X ⟵ a, Y ⟵ b, Z ⟵ v} = q(a,a,b,W,v)
q(a,a,b,W,v) {X ⟵ a, Y ⟵ b, Z ⟵ v} = q(a,a,b,W,v)

  → If substitutions are **impure**
    → Applying the substitution once
    → leads to an expression with an X
    → allowing for a different answer when the substitution is applied a second time
    →

q(X,X,Y,W,Z) {X ⟵ a, Y ⟵ b, Z ⟵ X} = q(a,a,b,W,X)
q(a,a,b,W,X) {X ⟵ a, Y ⟵ b, Z ⟵ X} = q(a,a,b,W,a)

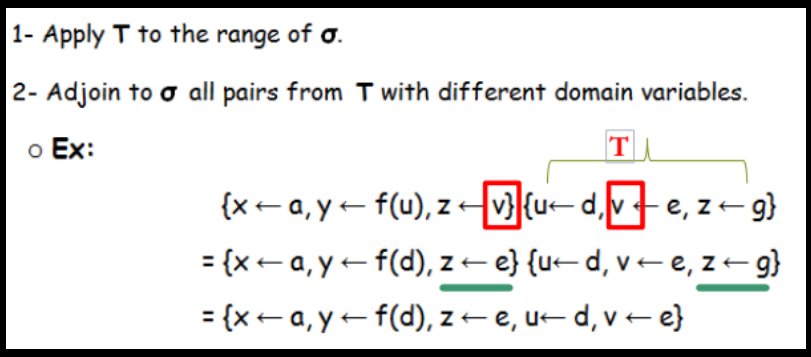
→ Types
  → Variable to Constant: X ← a
  → Variable to Variable: X ← y
  → Variable to Function: X ← f(Y)

Composition of Substitutions

→ the process of combining two or more substitutions into a single substitution that has the same effect as applying them in sequence.

→ Steps:

1. Apply the second substitution to the range of the first substitution
2. Adjoin all pairs from the second substitution that have different domain variables.



→ The composition of two impure substitutions may be pure, and vice versa

→ Composability

→ if the domain of the first and the range of the second are disjoint (i.e., they don't overlap).

→ If they overlap, the substitutions are non-composable.

→ Composability ensures preservation of purity

→ Composition of Two Impure Substitutions Leading to a Pure

Consider the substitution:

- $\alpha : z \mapsto f(z)$ (impure substitution, since $f(z)$ contains z)
- $\beta : z \mapsto g(z)$ (impure substitution, since $g(z)$ contains z)

Scenario: Composition of the Two Substitutions

Let's say we apply the substitution α first, followed by β.

Step 1: Apply α

When we apply $\alpha : z \mapsto f(z)$, we replace every occurrence of z in a given expression with $f(z)$. This results in an expression that may still contain z inside $f(z)$, so it's impure.

Step 2: Apply β after α

Now, we apply $\beta : z \mapsto g(z)$. In this step, we replace every occurrence of z in the expression (which, after applying α, could be inside $f(z)$) with $g(z)$.

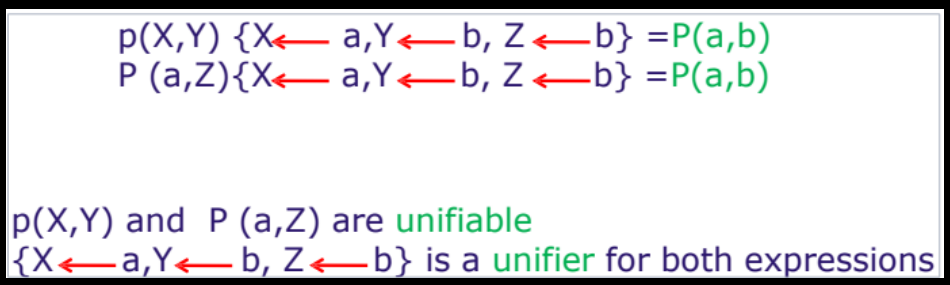
Now, we have $f(g(z))$. Notice that $g(z)$ inside $f(g(z))$ doesn't introduce any new free variables. Importantly, z is now **bound** within $g(z)$ and is not free in the expression anymore.

→ A **bound variable** → is a variable that is inside the scope of a function or a quantifier

Unifier

→ is a substitution that makes two expressions identical.

→ If two expressions have a unifier, they are said to be unifiable
otherwise, they are non-unifiable



→ Unification with Predicates

→ Constant to Free Variable:

→ by substituting the free variable with the constant.

→ Example: Loves(X, Y) and Loves(a, b) unify to X ← a and Y ← b.

→ Free Variable to Free Variable:

→ by substituting one variable for the other.

→ Example: Loves(X, Y) and Loves(Z, W) unify to X ← Z and Y ← W.

→ Constant to Constant:

→ can only be unified with each other if they are identical

→ Example: Loves(a, b) and Loves(a, b) unify successfully, but Loves(a, b) and Loves(a, c) do not.

Converting FOL to CNF --> INSEADOR-Steps

1. Implication Out:

→ implications (⇒⇒) and biconditionals (⇔⇔)

→ by replacing them with equivalent expressions

→ using ¬¬, ∧∧, and ∨∨

→ EX:

$$\varphi_1 \Longrightarrow \varphi_2 \longrightarrow \neg\varphi_1 \vee \varphi_2$$

$$\varphi_1 \Longleftrightarrow \varphi_2 \longrightarrow (\neg\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \neg\varphi_2)$$

2. Negations in
   - → Move negations inward
   - → using De Morgan's laws until they apply directly to atomic sentences.
   - → EX:

$$\begin{array}{lcl} \neg\neg\varphi & \rightarrow & \varphi \\ \neg(\varphi_1 \wedge \varphi_2) & \rightarrow & \neg\varphi_1 \vee \neg\varphi_2 \\ \neg(\varphi_1 \vee \varphi_2) & \rightarrow & \neg\varphi_1 \wedge \neg\varphi_2 \\ \neg\forall v.\varphi & \rightarrow & \exists v.\neg\varphi \\ \neg\exists v.\varphi & \rightarrow & \forall v.\neg\varphi \end{array}$$

3. Standardize Variables
   - → Rename variables so that each quantifier has a unique variable.
   - → EX:

$$\forall x[p(X)] \Rightarrow \exists x[q(X)] == \forall x[p(X)] \Rightarrow \exists y[q(Y)]$$

4. Existentials out
   - → drop existential quantifiers
   - → replacing them with Skolem constants or Skolem functions
   - → EX:

e.g., $\forall x[p(X) \wedge \exists y\ q(X,Y)] == \forall x[p(X) \wedge q(X,f(X)]$

e.g., $\forall x[p(X) \wedge \exists y\ q(X,Y,Z)] == \forall x[p(X) \wedge q(X,f(X,Z),Z]$

- **f(x,z),** because **y** depends on both the universally quantified variable **x** and the free variable **z**.
- **Skolem function** typically uses the arguments from the **same context** where the **existential** quantifier appears.

5. Alls out
   - → Drop all universal quantifiers, as the remaining variables are universally quantified.
   - → EX:

$$\forall x[p(X) \wedge q(X,Y,Z)] == p(X) \wedge q(X,Y,Z)$$

6. Distribution
   - → Distribute ∨∨ over ∧∧ to convert the formula into a conjunction of disjunctions.
   - →

$$\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \longrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$$

7. Operators
   → Separate conjunctions into individual clauses.
   →

$$\varphi_1 \vee \ldots \vee \varphi_n \longrightarrow \{\varphi_1, \ldots, \varphi_n\}$$
$$\varphi_1 \wedge \ldots \wedge \varphi_n \longrightarrow \{\varphi_1\} \ldots \{\varphi_n\}$$

8. Rename Variables
   → Rename variables so that no variable appears in more than one clause
   →

$$p(x) \longrightarrow p(x)$$
$$q(x) \longrightarrow q(y)$$

→ Example:

(1)

Convert the following FOL sentence into CNF
∃y ∀x p(X,Y)

I	∃y ∀x p(X,Y)
N	∃y ∀x p(X,Y)
S	∃y ∀x p(X,Y)
E	∀x p(X,a)
A	p(X,a)
D	p(X,a)
O	{p(X,a)}
R	{p(X,a)}

(2)

Convert the following FOL sentence into CNF
∀x ∃y p(X,Y)

I	∀x ∃y p(X,Y)
N	∀x ∃y p(X,Y)
S	∀x ∃y p(X,Y)
E	∀x p(X, g(X))
A	p(X, g(X))
D	p(X, g(X))
O	{p(X, g(X))}
R	{p(X, g(X))}

(3)

Convert the following FOL sentence into CNF

$\exists x\ \exists y[p(X,Y) \wedge q(X,Y)]$

I	$\exists x\ \exists y[\ p(X,Y) \wedge q(X,Y)]$
N	$\exists x\ \exists y[\ p(X,Y) \wedge q(X,Y)]$
S	$\exists x\ \exists y[\ p(X,Y) \wedge q(X,Y)]$
E	$p(a,b) \wedge q(a,b)$
A	$p(a,b) \wedge q(a,b)$
D	$p(a,b) \wedge q(a,b)$
O	$\{p(a,b)\}$ $\{q(a,b)\}$
R	$\{p(a,b)\}$ $\{q(a,b)\}$

(4)

Convert the following FOL sentence into CNF

$\neg\exists y.(g(y) \wedge \forall z.(r(z) \Rightarrow f(y,z)))$

I $\neg\exists y.(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z)))$

N $\forall y.(\neg(g(y) \wedge \forall z.(\neg r(z) \vee f(y,z)))$

$\forall y.(\neg g(y) \vee \neg\forall z.(\neg r(z) \vee f(y,z)))$

$\forall y.(\neg g(y) \vee \exists z.\neg(\neg r(z) \vee f(y,z)))$

$\forall y.(\neg g(y) \vee \exists z.(\neg\neg r(z) \wedge \neg f(y,z)))$

$\forall y.(\neg g(y) \vee \exists z.(r(z) \wedge \neg f(y,z)))$

S $\forall y.(\neg g(y) \vee \exists z.(r(z) \wedge \neg f(y,z)))$

E $\forall y.(\neg g(y) \vee (r(k(y)) \wedge \neg f(y,k(y))))$

A $\neg g(y) \vee (r(k(y)) \wedge \neg f(y,k(y)))$

D $(\neg g(y) \vee r(k(y))) \wedge (\neg g(y) \vee \neg f(y,k(y)))$

O $\{\neg g(y), r(k(y))\}$

$\{\neg g(y), \neg f(y,k(y))\}$

Note: **k** is a **Skolem** function

Predicate Resolution

→ is a rule of inference used to derive new clauses from existing ones.

→ Steps

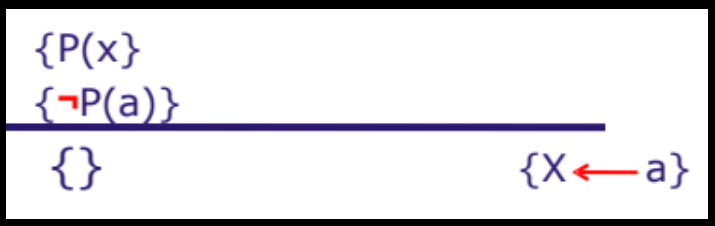
1. Resolution Process
   - → Finding Complementary Literals
   - → Look for literals in two clauses that are negations of each other
   - → EX:
     - → **P(X) and ¬P(a),** which contain **P and ¬P**

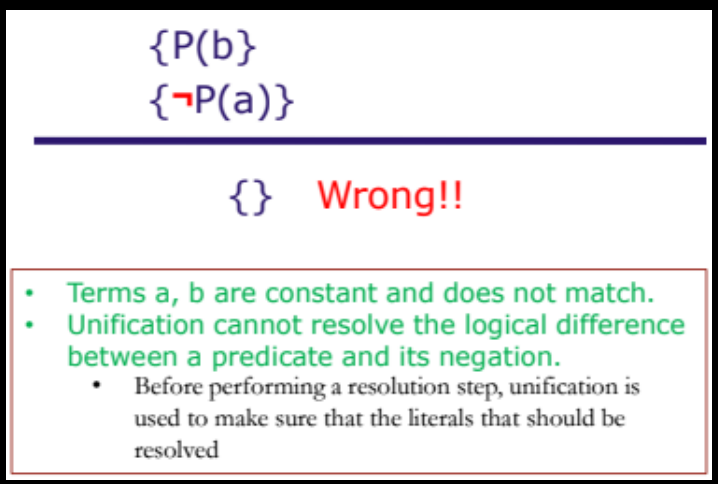
2. Unification
   - → Find a substitution that makes the complementary literals identical
   - → EX:
     - → **P(X) and ¬ P(a)** can only be unified if **X = a**

3. Performing Resolution
   - → Remove the complementary literals
   - → and combine the remaining parts of the clauses.
   - → EX:
     - → After removing **P(a) and ¬P(a),** nothing remains (**empty clause**)

4. Empty Clause
   - → If the result is an empty clause
   - → it indicates a contradiction
   - → meaning the original set of clauses is unsatisfiable

→ Resolvent





{P(X,Y)}
{¬P(c, b)}

{} {X ⟵ c, Y ⟵ b}

P(X, Y) and ¬P(c, b) are negations of each other. For the unification to work, we'd need to find a substitution that makes these literals the same, which is impossible because one is positive (P) and the other is negative (¬P).

{P(X,Y),q(X,g)}
{¬P(c, b)}

{q(c,g)} {X ⟵ c, Y ⟵ b}

{X ⟵ a, Y ⟵ b, Z ⟵ c, W⟵d}

{P(X,Y),q(Z,W)}
{¬P(a, b),¬q(c,d)}

{} Wrong!!

only one pair of literals may be resolved at a time

{X ⟵ a, Y ⟵ b, Z ⟵ c, W⟵d}

{P(X,Y),q(Z,W)}
{¬P(a, b),¬q(c,d)}

{q(Z, W), ¬q(c, d)}
{p(X, Y), ¬p(a, b)}

more than one resolvent

- The **first** resolvent from the first set of clauses is {q(Z, W), ¬q(c, d)}.
- The **second** resolvent results in an empty clause ({}), indicating a contradiction, and hence, no solution can be derived from the given clauses.

→ Example

(1)

Given the premises ∀x[p(X) ⇒ q(X)] and ∀x[q(X) ⇒ r(X)] , use resolution to prove the conclusion ∀x[p(X) ⇒ r(X)]

1) {¬p(X), q(X)}	Premise
2) {¬q(Y), r(Y)}	Premise
3) {P(a)}	Negated goal
4) {¬r(a)}	Negated goal
5) {¬q(a)}	2,4 {Y/a}
6) {¬p(a)}	1,5 {X/a}
7) {}	3,6

The **unifier** is **{Y/a, X/a}**

(2)

suppose we have the premises:

Art is the father of Jon

Bob is the father of Kim

Fathers are Parents

F(art, jon)
F(bob, Kim)
∀x ∀y [F(X,Y) → P(X,Y)]
P(art , jon)

We want to prove that , Art is a parent of Jon

1) {**F**(art,jon)}	Premise
2) {**F**(bob,kim)}	Premise
3) {¬**F**(X,Y),**P**(X,Y)}	Premise
4) {¬**P**(art, jon)}	Negated goal
5) {**P**(art, jon)}	1,3 {X/art, Y/jon}
6) {}	4,5

(3)

Another type of FOL resolution examples is (fill in blank)

For example, to ask about Jon's parent, we would write the Question p (Z, jon)

1) {F(art,jon)}	Premise
2) {F(bob,kim)}	Premise
3) {¬F(X,Y),P(X,Y)}	Premise
4) {¬P(Z, jon), goal(Z)}	Negated goal
5) {¬F(Z, jon), goal(Z)}	3,4 {X/Z, Y/jon}
6) {goal(art)}	1,5 {Z/art}

Which mean that the answer is art

Lecture (7)

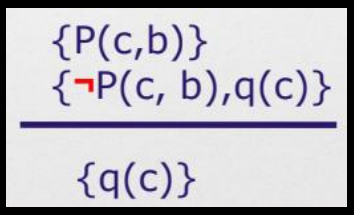
Horn Clause

→ a clause with at most one positive literal.

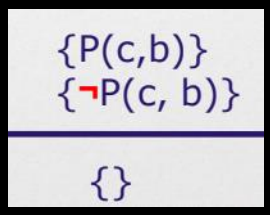
→ categories:
- → Fact:
  - → (1 positive, 0 negative)
  - → EX: {Father(ali, ahmed)}
- → Negated goal:
  - → (0 positive , 1 or more negative)
  - → EX: {¬ male(X), ¬ Father(ali,X)}
- → Null clause:
  - → (0 positive , 0 negative)
  - → EX: { } the clause appears at the end of resolution proof
- → Rule:
  - → (1 positive , 1 or more negative)
  - → EX: {¬ father(X,Y), parent(X,Y)}

→ The resolvent of two Horn clauses is a Horn clause
EX:

(1)



(2)



- → Prolog adopt the horn clause resolution where:
- → follows the **set of support strategy**
  - → No two clauses from the premises are resolved together.
  - → Every resolution step involves a negated goal and a fact or rule from the premises.

Prolog

- Imperative Languages:
  - also called procedural languages
  - It is about how the problem be solved.
  - Programmer gives all steps to solve a problem.
  - He must know an algorithm to solve a problem
  - EX:

Example: find average of list of numbers:
» Input numbers
» Compute total
» Average = total / numbers_count
» Print Average

- Declarative Languages
  - What is the problem?
  - describe the problem without the control flow
  - then the system will solve the problem.
  - Programmer must know the relations between objects to solve the problem.
  - Need not to know an algorithm to solve the problem.

- Prolog Program Sections
  - Sections
    - Database
      - To declare data base contain set of facts
    - Domains:
      - To declare domains of the variables
      - Giving meaningful names to domains
      - Declare data structures that don't defined by the standard prolog domains
      - **Char** → character →'a' - '/' - '&' - '3'
      - **Real** → decimal value → 2.4, 3.0, 5, -2.67, ...
      - **Integer** → numerical value → 1, 20, 0, -3, -50
      - **Symbol Or string** → a sequence of characters
        → Ex : telephone_number "Railway ticket"

- Clauses:
  - To declare constant values
  - Types:
    - Facts
      - Relation → Ayman plays Football
      - Properties → Sky is blue
    - Rules
      - enable you to infer facts from other facts
      - two parts Head and Body

        Head :- <subgoal$_1$>,<subgoal$_2$> , … , <subgoal$_N$>

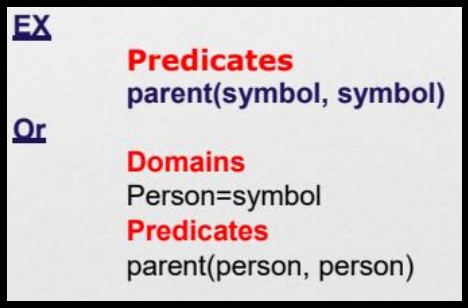
,	It mean Conjunction of sub-goals, all rules must be satisfied
;	It means Disjunction of sub-goals, any sub-goal provide rule satisfied
:-	It means(if) placed between rule head and rule body
.	it means the termination of Fact,Rule or Goal

      - Comments
        - Multiple line /**/
        - single line %

- Constants:
  - declare symbolic constants
  - indicated by the keyword constants
  - syntax: <id> = <value>
  -
    ```
    constants
            pi=3.14159
    ```

- Predicates:
  - To declare the predicates
  - predicate_name (arg_typel , arg_type2, … , arg_typeN).
    - **Names:**
      - must begin with letter
      - is not followed by period.
      - followed by a sequence of letters, digits and underscores
    - **Arguments:**
      - must belongs to known prolog standard domains
      - or one you declare in the domains section
      - can be equal to zero

- EX:



- Goal:
  - External goal
    - Written outside the program
    - Can be changed each time the program is executed
  - Internal goal
    - inside the program
    - Run automatically when the program is executed
    - Can't be changed during execution.

- Variables
  - must begin with a
    - capital letter
    - or an underscore
  - followed by any number of
    - letters
    - digits
    - underscores
  - get their values by being matched to constant in facts or rule (**Unification process**)
  - cannot store information by giving a value to a variable.
  - used to express
    - general rules
    - and to ask questions.
  - anonymous variables
    - represented by single underscore (_)
    - used to ignore the values
    - never get a value
    - match anything.

→ Examples

(1)

```
domains
Person= symbol
predicates
male(person)
father(person, person)
brother(person, person)
clauses
male(ali).
male(ahmed).
male(sleem).
father(sleem, ali).
father(sleem, ahmed).
brother(X,Y):- father(P,X),
               father(P,Y),
               male(X), male(Y).
```

goal

father(X,Y)

Answer

X=sleem, Y= ali
X=sleem, Y= ahmed

goal

father(_,Y)

Answer

Y= ali
Y= ahmed

goal

brother(X,Y)

Answer

X= ali
Y= ahmed

goal

male(X)

Answer

X=ali
X=ahmed
X=sleem

goal

male(ali)

Answer

yes

goal

male(ali) , male(muhammad)

Answer

No

(2)

```
domains
Person= symbol
food= symbol
predicates
likes(person, food)
clauses
likes(ali, apple).
likes(ali,pizza).
likes(omr, apple).
```

Compound goal:

You can use conjunction or disjunction in your query

goal

likes(ali,X), likes(omr,X)

Answer

X=apple

→ Arithmetic Operations

Addition	+
subtraction	-
multiplication	*
division	/
Integer part of division	div
Remainder of division	mod

→ Logical Operations

Greater than	>
Less than	<
Equal	=
Not equal	<>
Greater than or equal	>=
Less than or equal	<=

→ Compound objects

→ allow you to treat several pieces of information as a single item

→ you can easily pick them apart again

```
Domains
        date_cmp=myDate(string, integer, integer)
Predicates
        Person(string, date_cmp)
Clauses
        Person(ali, myDate("October",4,1993)).
```

→ functor

→ is just a name that identifies a kind of compound data object

→ hold its argument together

Lecture (8)

Unification & backtracking

- Unification:
  - when prolog tries to fulfill a goal
    - match the presence of the predicate itself
    - its arity and the types of these arguments
  - search from the top of the CLAUSES section
  - When it finds a clause that matches the goal
    - it binds values to free variables (substitution)
- Backtracking:
  - Finding all possible solutions by considering alternative paths
  - Prolog use backtracking

```
predicates
likes(symbol,symbol)
tastes(symbol, symbol)
food(symbol)
clauses
food(apple).
food(pizza).
tastes(pizza,good).
tastes(apple,bad).
likes(ali,X) :- food(X),
          tastes(X,good).
```

goal

```
Likes(ali, Pizza)
```

goal

```
likes(ali,X)
```

Answer

```
X= pizza
```

- Controlling the Search for Solutions
  - Fail
    - force backtracking
    - a false statement
    - corresponds to the effect of the comparison 2 = 3 or any other impossible sub-goal

```
predicates
father (symbol,symbol)
print
clauses
father(esa,sara).
father(samir,omr).
father(samir,hager).

print:- father(X,Y), write( X, "is
the father of",Y,"\n"), fail.
```

(Finally)

Output

```
esa is the father of sara
samir is the father of omr
samir is the father of hager
```

→ cut (!)

→ prevent backtracking

→ main uses

→ Green Cut: When you know it's a waste of time to look for alternate solutions (no meaningful solutions)

→ Red Cut: the logic of a program demands the cut

→ EX:

(1)

```
predicates
num (integer)
clauses
num(5).
num(7).
num(10).
```

```
goal
num(X)
```

Output

```
X=5
X=7
X=10
```

```
predicates
num (integer)
clauses
num(5):-!.
num(7).
num(10).
```

```
goal
num(X)
```

Output

```
X=5
```

(2)

```
predicates
num (integer)
letter(char)
Print(integer, char)
clauses
num(10).
num(20).
letter('a').
letter('b').
Print(X,Y):-num(X),letter(Y).
```

```
goal
print(X,Y)
```

Output

```
X=10    Y='a'
X=10    Y='b'
X=20    Y='a'
X=20    Y='b'
```

```
predicates
num (integer)
letter(char)
Print(integer, char)
clauses
num(10).
num(20).
letter('a').
letter('b').
Print(X,Y):-num(X),letter(Y),!.
```

```
goal
print(X,Y)
```

Output

```
X=10    Y='a'
```

```
predicates
num (integer)
letter(char)
Print(integer, char)
clauses
num(10).
num(20).
letter('a').
letter('b').
Print(X,Y):-num(X) ,!,letter(Y).
```

```
goal
print(X,Y)
```

Output

```
X=10    Y='a'
X=10    Y='b'
```

(3)

Prolog Code

```
op(X,Y,Z,O):-    O='+',Z=X+Y, !.
op(X,Y,Z,O):-    O='-',Z=X-Y, !.
op(X,Y,Z,O):-    O='*',Z=X*Y, !.
op(X,Y,Z,O):-    O='/',Z=X/Y, !.
op(X,Y,Z,_):-    write("Invalid").
```

C# Code

```
switch(O)
{
  case '+' : Z=X+Y; break;
  case '-' : Z=X-Y; break;
  case '*' : Z=X*Y; break;
  case '/' : Z=X/Y; break;
  default: console.wrtie("Invalid");
}
```

→ not Predicate

→ not predicate → succeeds when the sub-goal can't be proven true.

→ Free variables not allowed in "not"

→ EX:

(1)

```
predicates
animal(symbol)
has_wings(symbol)

clauses
animal(dog).
animal(cat).
animal(bird).
has_wings(bird).
```

goal

```
not(has_wings(bird)).
```

Output

No

(2)

```
predicates
animal(symbol)
has_wings(symbol)

clauses
animal(dog).
animal(cat).
animal(bird).
has_wings(bird).
```

goal

```
not(has_wings(cat)).
```

Output

Yes

(3)

```
predicates
animal(symbol)
has_wings(symbol)

clauses
animal(dog).
animal(cat).
animal(bird).
has_wings(bird).
```

goal

```
Animal(X),
not(has_wings(X)).
```

Output

```
X=dog
X=cat

2 solutions
```

Wrong

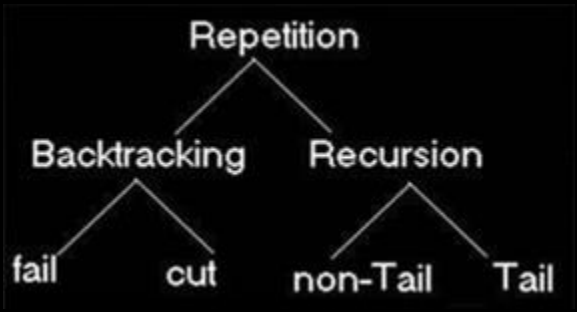
goal

```
not(has_wings(X)), Animal(X).
```

Free variable is not allowed in 'not'

Lecture (9)

Repetition



Recursion

- Any recursion contains the steps:
  - Base case
  - Recall
- EX:

  Non-Tail Recursion:

EX: **Factorial**

To find the factorial of a number X

Base case:
If X is 1, the factorial is 1

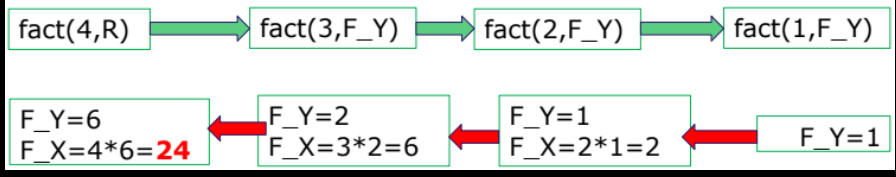
Recall:
find the factorial of X-1, then multiply it by X

```
Predicates
 fact (integer, real)
Clauses

fact(1,1).

fact(X, F_X) :-
      Y=X-1,
      fact(Y, F_Y),
      F_X =X * F_Y.
```

```
Predicates
 fact (integer, real)
Clauses
    fact(1,1).
    fact (X, F_X) :-
        Y=X-1,
        fact(Y, F_Y),
        F_X =X * F_Y .
```

goal
fact(4,R)
R=24

Recursion big drawback:
It eats memory.
Solution
tail recursion optimization



Tail Recursion:

- The call is the very last sub-goal of the clause
- no backtracking points

→ It save memory (no backward direction)

```
Predicates
 count(integer)
Clauses

    count(1001):-!.

    count(X):-    write(X), nl ,Y=X+1, count(Y).
```

```
goal
        count(0)
```

program counts from 0 to 1000

Not a Tail Recursion

```
Predicates
 count(integer)
Clauses

    count(1001):- !.

    count(X):-  write(X),Y=X+1, count(Y), nl.
```

```
Predicates
 count(integer)
Clauses

    count(1001):- !.

    count(X):-  write(X),Y=X+1,nl, count(Y).
    count(X):-  X<0, write("negative").
```

List processing

- → an object that contains an arbitrary number of other objects within it
- → does not require you to declare how big it will be before you use it
- → Declaring Lists
  - → Domains
    - → list_name = data_type *
  - → consists of
    - → Head: the first element
    - → ail:

→ all the subsequent elements.

→ The tail of a list is always a list

→ EX:

List	Head	Tail
['a' , 'b' , 'c']	'a'	['b' , 'c']
['a']	'a'	[ ]
[ ]	Undefined	Undefined
[[1,2,3] , [2,3,4] , []]	[1,2,3]	[[2,3,4], [ ]]

→ separate the head and tail with vertical bar "|"

List_1	List_2	Variable binding
[X , Y , Z]	["omar","eats", "icecream"]	X=omar Y=eats Z=icecream
[1, 2,3]	[X|Y]	X=1 Y=[2,3]
[1,2,3,4]	[X,Y|Z]	X=1 Y=2 Z=[3,4]

→ **Examples:**

(1) Print list elements

Base case:
To print empty list, do nothing.

Recall:
to print list elements:
- print its head,
- then print tail (a list)

```
DOMAINS
list = integer*

PREDICATES
print (list)

CLAUSES
print( [ ] ) :-!.
print( [ H |T ] ) :-
                write (H) ,nl,
                print (T).
```

Goal: print([1,20,45])

(2) Search for an element

→ Could be the first element of the list

→ Or a member of the its tail

```
DOMAINS
list = integer*
PREDICATES
member(integer, list)
CLAUSES
member(X, [H | T ] ):- X=H.
member(X, [ H | T ]) :- X<>H, member (X, T).
```

Warning: The variable is only used once.
Solution: replace with _

```
goal
        member(5,[1,2,3,4])
```

No

```
goal
        member(4,[1,2,3,4])
```

Yes

```
DOMAINS
list = integer*
PREDICATES
member(integer, list)
CLAUSES
member(X, [ ]):- write(X," is not found "), fail.
member(X, [X | _ ] ):- write(X, " is found ").
member(X, [ H | T ]) :- X<>H, member (X, T).
```

```
goal
        member(5,[1,2,3,4])
```

5 is not found
No

```
goal
        member(4,[1,2,3,4])
```

4 is found
Yes

(3) Copy list to another

→ While List_1 is not empty, use recursion to transfer one element at a time to List_2

→ When List_1 is empty add nothing to list_2 tail

```
DOMAINS
list = integer*
PREDICATES
copy(list ,list)

CLAUSES
copy( [],[]).
copy( [H|T1] ,[H2|T2] ) :- H2=H, copy(T1,T2).
```

Tracing:
copy([1,2,3,4],[]) X=[]
copy([2,3,4], []) X=[1]
copy([3,4], []) X=[1,2]
copy([4], []) X=[1,2,3]
copy([], []) X=[1,2,3,4]

```
Goal:   copy([1,2,3,4],X)
```

X=[1,2,3,4]
1 solution

(4) What is the output?

```
DOMAINS
list = integer*
PREDICATES
copy(list ,list)

CLAUSES
copy( [],[]) .
copy( [H|T1] ,[H,H|T2] ) :- copy(T1,T2).
```

Goal: copy([1,2,3,4],X)

X=[1,1,2,2,3,3,4,4]

```
DOMAINS
list = integer*
PREDICATES
copy(list ,list)

CLAUSES
%copy( [],[]) .
copy( [H|T1] ,[H,H|T2] ) :- copy(T1,T2).
```

Goal: copy([1,2,3,4],X)

No Solution

(5) Sum of List: by recursively processing the elements

Non-tail recursion

```
DOMAINS
list = integer*
PREDICATES
sum_list(list ,integer)
CLAUSES
Sum_list([ ], 0).          % Base case: the sum of an empty list is 0.
sum_list( [H|T], Sum) :-
   sum_list( T, TailSum),  % Recursive call on the tail of the list.
   sum= H + TailSum.       % Add the head element to the sum of the tail.
```

Goal: sum_list([1,2,3,4],X) X = 10

Tail recursion

Using Tail-Recursion

```
DOMAINS
list = integer*
PREDICATES
sum_list2(list ,integer)
sum_list2(list ,integer, integer)

CLAUSES
sum_list2(List, Sum) :-
   sum_list2(List, 0, Sum).   % Call the helper predicate with an initial accumulator of 0.

% Tail-recursive helper predicate
sum_list2([ ], Acc, Acc).     % When the list is empty, return the accumulator.
sum_list2([H|T], Acc, Sum) :-
   NewAcc=Acc + H,            % Add the head to the accumulator.
   sum_list2(T, NewAcc, Sum). % Recurse on the tail with the updated accumulator.
```

Goal: sum_list2([1,2,3,4],X) X = 10

(6) Collect list elements at once

→ Findall

→ takes a goal as one of its arguments

→ and collects all the solutions to that goal into a single list

→ **three arguments:**

→ Var_name: argument to be collected into a list

→ Predicate_name: predicate from which values are collected

→ List_Param: list of values collected through backtracking

```
PREDICATES
Person(symbol, integer)

CLAUSES
Person("ali", 20).
Person("omar", 28).
Person("ali", 25).
```

Goal

```
findall(Age, person(_, Age), L)
```

L=[20,28,25]

(7) Combining AND and OR

→ Create a separate rule for the ORing subgoal

→ Include the ORing rule within the ANDing rule

Clauses

```
subgoal(1).
subgoal(2).
subgoal(3).
subgoal(4).

or_goal :- subgoal(1); subgoal(2).
parent_goal :- subgoal(3), subgoal(4), or_goal.
```