**Unit 1**

 **Introduction to Logic**

**What is Logic**? Logic is about the relationship among proposition. Logic is an attempt to understand when one statement follows from other statements. Logic is about what follows from what; it's about how to tell good arguments from bad arguments.

Thus, Logic is concerned with reasoning and the validity of arguments. In general, in logic, we are not concerned with the **truth**of statements, but rather with their **validity**. That is to say, although the following argument is clearly logical, it is not something that we would consider to be true:

All lemons are blue

Mary is a lemon

Therefore, Mary is blue

This set of statement is considered to be **valid** because the conclusion (Mary is blue) follows logically from the other two statements, which we often call the **premises**.

Logic is also concerned with **truth values**. The possible truth values are true and false. These can be considered to be the fundamental units of logic, and almost all logic is ultimately concerned with these truth values.

* 1. **Historical view of Logic**
* **Philosophical Logic /symbolic logic:** **The First Age of Logic**
	+ 500 BC to 19th Century
	+ Logic dealt with arguments in the natural language used by humans. Example
	+ All men are mortal.
	+ Socrates is a man
	+ Therefore, Socrates is mortal.

If we change “all’ to “some”, the argument doesn't hold. But how could we demonstrate this? We could attempt to dene words such as “all” and “some” in terms of what inferences we could draw from them. Then the demonstration would follow from these definitions. The problem is that natural language turns out to be very ambiguous.

* **The 2nd Age of Logic : Algebraic Logic**
	+ Mid to late 19th Century
	+ Attempted to formulate logic in terms of a mathematical language
	+ Rules of inference were modeled after various laws for manipulating algebraic expressions.
* **Mathematical Logic: The 3rd Age of Logic**
	+ Late 19th to mid-20th Century
	+ Stated proposed logic as a language for mathematics in 1879.
	+ Russell’s Paradox concluded that this leads to the so- called Russell's Paradox: Consider the set T = {S | S ∉ S} then T € T ⇔T ∉T (a contradiction).
* **Logic in Computer Science: The 4th Age of Logic**

In computer science, we design and study systems through the use of formal languages that can themselves be interpreted by a formal system.

* + Boolean circuits
	+ Programming languages
	+ Design Validation and verification
	+ AI: mechanized reasoning and expert systems.
	+ Security: Concept of proof- carrying code. Etc.
* Logic in Computer Science includes the following issues: There are many logics.
	+ Propositional Logic
	+ Predicate logic: First Order Logic, Higher Order Logic
	+ Theory of Construction
	+ Real-time Logic, Temporal Logic
	+ Process Algebras
	+ Linear Logic
	+ Etc

Therefore, hundreds of logics have been studied by philosophers, computer scientists and mathematicians. Logic has been called “**the calculus of computer science**”.

* 1. **Aim of Logic in Computer Science**
* The aim of logic in computer science is to develop languages to model the situations with reasoning.
* Reasoning about situations does mean constructing arguments about them for validity.
* For propositional logic, models are assignments of true or false to every proposition symbol. Thus, logic could be defined as many ways:
* It is a foundation for organized and careful method of thinking that characterizes reasoned activity.
* It is the study of reasoning: specifically concerned with whether something is correct or incorrect.
* Since Logic is involved in broad range of intellectual activities and it is a base in many areas of computer science such as artificial intelligence, algorithms etc., the study of logic is essential for the computer science.
1. Logic has been an effective tool for answering some of the basic questions in complexity. Examples: NP-complete problems.
2. It is also proven that Logic is a spectacularly effective tool in databases. For example, In SQL query can be broken down to sub queries where we use First Order Logic to break the Query into Sub queries.
3. Reasoning about Knowledge.
	1. **Application of Logic in Computer Science**

In general the applications of logic in computer science are:

* + - Circuit design
		- Boolean search
		- Program specification
		- Program verification
		- Logic programming
		- Automated theorem-proving
		- Database Theory
		- Design Verification
		- Security
		- Programming Languages and Semantics
		- AI Reasoning, and Common Sense reasoning,
		- Software engineering (providing specifications to programmers) etc.
	1. **Propositions / Statements**
* What is a proposition or statement?

A statement / proposition is a declarative sentence (that is, a sentence that declares a fact) that is either true or false, but not both.

* Statements state a fact, or describe a state of relationships.
* **Examples**: the following are propositions
	+ The sum of the numbers 3 and 5 equals 8.
	+ Addis Ababa is the capital of Ethiopia.
	+ 2 + 2 = 3.
* The following are not propositions
	+ What time is it?
	+ Read this carefully.
	+ x + 1 = 2.
* Statements or propositions can be atomic (indivisible) or compound (made up of two or more atomic) propositions.
* Compound propositions are formed by combining individual propositions using logical operators or rules of inferences.
* Propositions are the building blocks of propositional logic.

Propositional Logic is a formal language consisting of symbols (propositional variables), logical operators (connectives) and inference rules for manipulating the symbols to construct a valid argument.

 **Chapter 2**

 **Propositional Logic**

* 1. **Definition of Propositional Logic**

Propositional Logic is also called **Propositional calculus** and /or **Boolean logic** after the logician George Boole (1815-1864). It isa very simple logic Propositional calculus (or logic) is the study of the logical relationship between objects called propositions and forms the basis of all mathematical reasoning.

The propositional calculus is based on statements which have truth values (true or false) and the logical connectives: **binary** connectives ( ‘and,’ ‘or,’ ‘imply’ ‘bi-imply’) and **unary** connective (‘not,’).

The calculus provides a means of determining the truth values associated with statements formed from “atomic” statements. An example:

If p stands for “Almaz is logical” and q for “Almaz is tall” then we may form statements such as:



* 1. **Syntax and Semantics of Propositional Logic**

A logical system can be defined in terms of its syntax (the alphabet of symbols and how they can be combined), its semantics (what the symbols mean), and a set of rules of **deduction** that enable us to derive one expression from a set of other expressions and thus make arguments and proofs.

Syntax refers to the formal notation for writing assertions. It also refers to the data structures that represent assertions in a computer. At the level of syntax, 1 + 2 is a string of three symbols, or a tree with a node labelled + and having two children labelled 1 and 2.

The **syntax** of propositional logic defines the allowable statements/sentences. The **atomic sentences-**the indivisible syntactic elements-consist of a single **proposition symbol.** Each such symbol stands for a proposition that can be true or false.

Syntax can be represented in a computer. Proof methods are syntactic, so they can be performed by computer. On the other hand, as semantics is concerned with meaning, it exists only inside people’s heads.

Semantics expresses the meaning of a formula in terms of mathematical or real world entities. While 1 + 2 and 2 + 1 are syntactically distinct, they have the same semantics, namely 3. The semantics of a logical statement will typically be true or false. Propositional Logic is a formal language. Each formula has a meaning (semantics) either t or f relative to the meaning of the propositional symbols it contains. The meaning can be calculated using the standard truth tables.

* 1. Logical Connectivity and Truth Table
		1. **Logical Connectors**

Logical connectives will allow us to build up complex propositions. Propositions and logical connectives arise all the time in computer programs. For example, consider the following snippet, which could be either C, C++, or Java: if ( x > 0 || (x <= 0 && y > 100) )

Connectives are used to create a compound proposition from two or more other propositions. The 5 basic connectives in logic are:

1. Logical Conjunction: ^ and (&)
2. Logical Disjunction: ˅ (or)­
3. Negation: ¬ not ( ~ )
4. Implication: If…then, implies
5. Biconditional ( “if and only if”): iff
6. **Logical Conjunction: ^ and (& or.):** Any two propositions can be combined to form a third proposition called the conjunction of the original propositions.
* When two propositions **P** and **Q** are joined by the connective “and”, the resulting statement is called a **logical conjunction** or simply **conjunction** and is denoted by **P∧Q** and is read as “**P and Q**”.
* If p and q are arbitrary propositions, then the conjunction of p and q is written as: “p ^ q” and will be true iff both p and q are true.

NB: Check the operation of ^ in a truth table.

1. **Logical Disjunction: ˅ or (| or +)­:** Any two propositions can be combined by the word ‘or’ to form a third proposition called the disjunction of the originals.
* When two propositions **P** and **Q** are joined by the connectives “or”, then the proposition formed is called a **logical disjunction** or simply a **disjunction** and is denoted by **P∨Q** and is read as “**P or Q**”.
* If p and q are arbitrary propositions, then the disjunction of p and q is written p ˅ q and will be true iff either p is true, or q is true, or both p and q are true.

NB: Check the operation of ˅ in a truth table.

1. **Negation: ¬ not (~):** unary connective that takes one argument.
	* + The negation of a statement is generally formed by introducing the word “**not**” at a proper place in a statement or by prefixing the statement with the phrase, “**It is not the case that**”
		+ Let P be a proposition. If P is modified by connective “not”, then the resulting proposition is called the *logical negation* of P denoted by ¬P.
* If p is an arbitrary proposition then the negation of p is written **¬**p and will be true iff p is false.

NB: Check the operation of **¬** in a truth table.

1. **Implication: , If…then, implies:** In propositional logic, we have a connective that combines two propositions into a new proposition called the conditional, or implication of the originals, that attempts to capture the sense of such a statement.
* When two propositions **P** and **Q** are joined by the connectives “**implies**” or “**if…then**”, the resulting statement denoted by **P⇒Q** is called an **implication** (**conditional**). In **P⇒Q**, **P** is called the **premise** or the **antecedent**; and **Q** is called the **conclusion** or the **consequent.**
* An implication or conditional of two simple propositions, written in the form **P⇒Q** has a truth value ‘false’ only if **P** is true and **Q** is false; otherwise it is true.
* If p and q are arbitrary propositions, then the conditional of p and q is written p q and will be true iff either p is false or q is true.

NB: Check the operation of  in a truth table.

1. **Biconditional ( “if and only if”): iff:**
* Two propositions **P** and **Q** are connected by the connective “if and only if” (*iff*) is denoted by **P⇔Q** is called a bi-implication or double implication.
* If p and q are arbitrary propositions, then the biconditional of p and q is written:

p q and will be true iff P and q are both true or P and q are both false.

If p q is true, then p and q are said to be logically equivalent. They will be true under exactly the same circumstances.

NB: Check the operation of  in a truth table.

**Precedence of Logical Operators**

 Just as in arithmetic, an ordering must be imposed on the use of logical operators in compound propositions.

1. (¬) Negation
2. (^) Conjunction
3. (˅) Disjunction
4. ( ) Implication
5. ( ) Biconditional

**2.3.2. Truth table:**

A truth table is a convenient format for displaying the semantics of a formula by showing its truth value for every possible interpretation of that formula. Therefore,

* Truth tables used to represent Truth values of propositions in a suitable form.
* Truth tables are tables presenting all the possible assignments of truth values of the component propositions and the derived complex propositions.
* In the initial columns of the truth table we write the possible truth values of the constituent statements and in the last column the truth value of the compound proposition is written.
* If a complex proposition is made up of ‘n’ simple propositions, then the number of rows will be 2n.

**Exercise:**

What are the truth values of the following Propositions?

1. If I live in Ethiopia, then I don’t speak Amharic.
2. If you go to Mexico, and I follow you, I won’t be able to find you.
3. Draw a truth table for the following expression: *￢*A *∧* (A *∨* B) *∧* (B *∨* C)
	1. **Propositional wffs**

In computer science, an *expression* denoted the computation of a value from other values; for example, 2 \* 9 + 5. In propositional logic (PL), the term *formula* is used instead.

Propositional calculus is the language of propositions (statements that are true or false).

We represent propositions by formulas called well-formed formulas (wffs) that are constructed by using the construction rules from an alphabet consisting of:

**Truth symbols**: T (True) and F (False)

**Propositional variables**: mostly in the uppercase letters. Every propositional atom *p, q, r, . . .* and *p*1*, p*2*, p*3*, . . .* is a well formed formula.

**Connectives (operators)**:

 ¬ (not, negation): If qis a well-formed formula, then so is (*￢q*).

∧ (and, conjunction)**:** If qand *p* are well-formed formulas, then so is (*q ∧ p*).

∨ (or, disjunction): If *q* and *p* are well-formed formulas, then so is (*q ∨ p*).

→ (conditional, implication):If *q* and *p* are well-formed formulas, then so is (*q → p*).

Parentheses symbols: (and).

On the other hand, Wffs are constructed using the following rules:

1. *True* and *false* are wffs.
2. Each propositional constant (i.e. specific proposition), and each propositional variable (i.e. a variable representing propositions) are wffs.
3. Each atomic formula (i.e. a specific predicate with variables) is a wff.
4. If *A, B,* and *C* are wffs, then so are *A*, (*A* *B*), (*A B)*, *(A B),* and *(A B)*.
5. If *x* is a variable (representing objects of the universe of discourse), and *A* is a wff, then so are *x A* and *x A* .

**Translating English to Wffs:**

To analyze reasoning, we have to be able to translate English to wffs. Consider the following sentences:

1. Chala doesn't love Hana.

2. Chala loves Hana and loves Martha

3. Chala loves Hana or Martha

4. Chala loves Hana but doesn't love Martha

5. If Chala loves Hana then he doesn't love Martha

First find the appropriate primitive propositions:

* P: Chala loves Hana
* Q: Chala loves Martha

Then translate:

1. ¬P

2. P ^ Q

3. P *∨* Q

4. P ^ ¬Q (note: “but” becomes “and”)

5. P *→* ¬Q

* 1. **Tautologies, Contradiction and Validity**
* What are tautology, contradiction and validity?

A wff of **PL** is a **tautology** if it takes the value true on every valuation of its atoms while a wff of **PL** is a **contradiction** if it takes the value false on every valuation of its atoms.

A tautology (or theorem) is a formula that evaluates to True for every truth assignment.

A compound proposition that is always true, no matter what the truth values of the propositions that occur in it is called a tautology. Whereas, a compound proposition that is always false is called a contradiction. In addition, a proposition that is neither a tautology nor a contradiction is called a contingency.

These concepts could be also stated as:

* A formula is a tautology iff it is true under every valuation;
* A formula is consistent iff it is true under at least one valuation;
* A formula is inconsistent iff it is not made true under any valuation.

Each line in the truth table of a formula corresponds to a valuation. A valuation is a function which assigns a truth value to each primitive proposition. So, we can use truth tables to determine whether or not formulae are tautologies.

Propositions p and q are logically equivalent if p ^ q is a tautology.

Evaluating Formulas

Given a formula, we want to decide if it is true or false. How do we deal with a complicated formula like:

P ^ (¬P) → (Q) *→* (R *∨*P)))

The truth or falsity of such a formula depends on the truth or falsity of the primitive propositions that appear in it. We use truth tables to describe how the basic connectives

(¬,^ , *∨*,*→* , ) work.

* 1. **Valid Arguments and Proofs**

An argument is logically valid if and only if its conclusion is a logical consequence of its premises. We'll say that an argument is valid if and only if it is impossible for its premises to be true and its conclusion false. This means that for all valid arguments, if the premises are true, then the conclusion has to be true. Another way to put this is to say that an argument is valid iff the truth of the premises guarantee the truth of the conclusion.

Obviously true premises with an obviously false conclusion are called **invalid** argument.

Definition of an argument has the form

A1,A2, ...An \_\_B

A1...An are called the premises of the argument; B is called the conclusion. An argument is valid if, whenever the premises are true, then the conclusion is true.

Theorem: An argument is valid if the conjunction of its premises logically implies the conclusion.

Proof: Suppose the argument is valid. We want to show

(A1 ^ ... ^ An) *→* B is a tautology.

NB:

1. The negation of a tautology is a contradiction.
2. The negation of a contradiction is a tautology.
3. If the conclusion of an argument is a tautology, then the argument is valid.

**Testing Arguments for Validity in PL**

Ex1. (1) Jack is logical.

 (2) It isn't the case that Jack is logical but Jill isn't.

 (3) Jill is logical.

Translation key:

Let, P = Jack is logical

 Q = Jill is logical

(1) P

(2) ¬ (P ∧ ¬Q)

(3) Q

Now validate with truth table as:



***Ex2***. (1) Either Jack or Jill went up the hill.

 (2) It isn't the case that Jack went up the hill and Jo didn't.

 (3) Hence it isn't the case that Jo went up the hill and Jill didn't.

***Translation key:***

Let P = Jack went up the hill

Q = Jill went up the hill

R = Jo went up the hill

(1) (P ∨ Q)

(2) ¬ (P ∧ ¬R)

(3) ¬(R ∧ ¬Q)

Now validate with truth table as:



For P ⇒ T, Q ⇒ F, R ⇒ T, the argument has all true premises and false conclusion.

So it is INVALID!

***Expressing Arguments in PL:***

• Use the symbol "," to separate premises in **PL**.

• Use the symbol "∴" to indicate a conclusion in **PL**.

For instance, we can rewrite the above exercises as:

***Ex1:*** P, ¬ (P ∧ ¬Q) ∴ Q

***Ex2***: (P ∨ Q), ¬(P ∧ ¬R) ∴ ¬(R ∧ ¬Q)

These strings of symbols are completely in the language **PL**.

• Technically, we could add another Rule of Grammar that precisely defines such "argument strings". (These strings are not *wffs*!)

An argument (string) in **PL** is ***tautologically valid*** just when its premises tautologically entail its conclusion.

***Ex3:*** (P ∨ Q), (R ∨ ¬P), (¬¬R ∨ ¬Q) ∴ ¬R



In the table, there is only one line where both premises are true, line 1. And in this line the conclusion is also true. That means it is indeed impossible for the premises to be true and the conclusion false -the truth table shows all possible situations. Thus the argument is valid.

***Ex4:*** (¬P ∨ R), (P ∨ Q), ¬(Q ∧ ¬S) ∴ (R ∨ S)

An Argument is:

**Valid** if the conclusion is true whenever the premises are true.

**Invalid** if it is possible for the premises to be true and the conclusion false.

Proof is another syntactical concept. A proof is a deduction of a formula from a set of formulas called axioms using rules of inference. The central theoretical result that we prove is the soundness and completeness of the axiom system: the set of provable formulas is the same as the set of formulas which are always true.

* 1. **Derivation Rules for Propositional Logic, Deduction Method and Additional Inference Rules**

A derivation (or proof) in an axiom system AX is a sequence of formulas

C1 ...CN; each formula Ck is either an axiom in AX or follows from previous formulas using an inference rule in AX:

i.e., there is an inference rule A1 ... An ` B such that Ai = Cji for some ji < N and B = CN.

This is said to be a derivation or proof of CN.

A derivation is a syntactic object: it's just a sequence of formulas that satisfy certain constraints.

\_ Whether a formula is derivable depends on the axiom system

\_ Different axioms! Deferent formulas derivable

\_ Derivation has nothing to do with truth!

How can we connect derivability and truth?

The logical equivalences below are important equivalences that should be memorized.



* 1. **Verbal Arguments**

A propositional language is based on two components, an alphabet and a grammar.

a) The alphabet consists of three sets:

i) a set of connective symbols { ¬, ... },

ii) a set of punctuation symbols {*(*, *)*},

iii) a set of propositional variables *P*.

b) The grammar, which defines what counts as being a sentence based on *P*:

i) each of the elements of *P* is a sentence based on *P*,

ii) if S and T are sentences based on *P* then so are (¬S), (S ^ T),

(S ∨ T), (S *→* T) and (S  T),

iv) nothing else is a sentence based on *P*.

 **Chapter 3**

 **Normal Forms**

The language of propositional logic is redundant: many of the connectives can be defined in terms of others. By repeatedly applying certain equivalences, we can transform a formula into a normal form. A typical normal form eliminates certain connectives entirely, and uses others in a restricted manner. The restricted structure makes the formula easy to process, although the normal form may be exponentially larger than the original formula. Most normal forms are unreadable, although Negation Normal Form is not too bad. Using logical equivalences, we can transform logical sentences into more standard forms that are important for applications of computers science. We will use the equivalences we saw earlier (e.g., DeMorgan’s laws, …)

Logical Operators: Do we need all these?

∨ - Disjunction

∧ - Conjunction

¬ - Negation

→ - Implication p→q ⇔ ¬p ∨ q

⊕ - Exclusive or (p ∧ ¬q) ∨ (¬p ∧ q)

↔ - Biconditional p ↔ q ⇔

 (p→q) ∧ (q→p) ⇔

 (¬p ∨ q) ∧ (¬q ∨ p)

The ∧, ∨, and ¬ form a functionally complete set of operators.

Are¬(p∨(¬p∧q)) and (¬p ∧ ¬q) equivalent?

¬(p∨(¬p∧q))

* + ¬p ∧ ¬(¬p∧q) DeMorgan
	+ ¬p ∧ (¬¬p∨¬q) DeMorgan
	+ ¬p ∧ (p∨¬q) Double Negation
	+ (¬p∧p)∨(¬p ∧¬q) Distribution
	+ (p∧¬p)∨(¬p ∧¬q) Commutative
	+ F ∨(¬p ∧¬q) And Contradiction
	+ (¬p ∧¬q) ∨ F Commutative
	+ (¬p ∧¬q) Identity

Two FOL sentences P and Q mean the same thing (are logically equivalent, written P ⬄ Q) iff they have the same truth value in all situations. If two sentences are logically equivalent, you can substitute one for the other. The identity laws are stated below.



Definition 2 (Normal Forms)

* A literal is an atomic formula or its negation. Let K, L, L0, ... stand for literals.
* A formula is in Negation Normal Form (NNF) if the only connectives in it are ^, v, and ¬ , where ¬ is only applied to atomic formula.

An atomic formula like P is in all the normal forms NNF, CNF, and DNF. NNF can reveal the underlying nature of a formula. For example, converting  to NNF yields A ^ ¬B. This reveals that the original formula was effectively a conjunction. Every formula in CNF or DNF is also in NNF, but the NNF formula  is in neither CNF nor DNF.

Every formula can be translated into an equivalent formula in NNF, CNF, or DNF

by means of the following steps.





* 1. **Disjunctive Normal Forms (DNF)**

Every compound proposition in the propositional variables p, q, r, ..., is uniquely equivalent to a proposition that is formed by taking the disjunction of conjunctions of some combination of the variables p; q; r; ... or their negations. This is called the disjunctive normal form of a proposition.

The disjunctive normal form of a compound proposition is a natural and useful choice for representing the proposition from among all equivalent forms, although it may not be the simplest representative.

A formula is in Disjunctive Normal Form (**DNF**) if it has the form A1 v… v Am, where each Ai is a conjunction of one or more literals.

**Example1**: Find the disjunctive normal form for the proposition p *→* q.

Solution: Construct a truth table for p *→* q:



p *→* q is true when either p is true and q is true, or p is false and q is true, or p is false and q is false.

The disjunctive normal form is then

 

This example shows how a truth table can be used in a systematic way to construct the disjunctive normal forms.

Example2: Construct the disjunctive normal form of the proposition

(P *→* q) ^¬r

Solution: Write out the truth table for (p *→* q) ^¬r:



The disjunctive normal form will be a disjunction of three conjunctions, one for each row in the truth table that gives the truth value T for (p *→* q) ^¬r. These rows have been boxed. In each conjunction we will use p if the truth value of p in that row is T and ¬p if the truth value of p is F, q if the truth value of q in that row is T and ¬q if the truth value of q is F, etc. The disjunctive normal form for (p *→* q) ^¬r is ; because each of these conjunctions is true only for the combination of truth values of p, q, and r found in the corresponding row. That is, (p ^q^¬r) has truth value T only for the combination of truth values in row 2, (¬p^q^¬r) has truth value T only for the combination of truth values in row 6, etc. Their disjunction will be true for precisely the three combinations of truth values of p, q, and r for which (p *→* q) ^¬r is also true.

* 1. **Conjunctive Normal Forms (CNF)**

A formula is in Conjunctive Normal Form (**CNF**) if it has the form A1 ^ ... ^ Am, where each Ai is a disjunction of one or more literals. The formula

is in CNF.

Unlike in some hardware applications, the disjuncts in a CNF formula do not have to mention all the variables.

For example, three applications of De Morgan's Laws gives 

Thus, if you want to get the conjunctive normal form of a proposition, construct the disjunctive normal form of its negation and then negate again and apply De Morgan's Laws.

Example3: Find the conjunctive normal form of the proposition (p^¬q) ∨r.

Solution:

(1) Negate: ¬ [(p ^ ¬q) ∨r] , (¬p ∨q) ^¬r:

(2) Find the disjunctive normal form of (¬p ∨ q) ^ ¬r:



The disjunctive normal form for (¬p ∨q) ^¬r is 

(3) The conjunctive normal form for (p^¬q) ∨r is then the negation of this last expression, which, by De Morgan's Laws, is

 **Chapter 4**

 **Predicate Logic**

A predicate is a proposition whose truth depends on the value of one or more variables. Propositional logic is not sufficiently expressive for formalizing mathematical theories such as arithmetic. An arithmetic expression such as x + 2 > y − 1 is neither true nor false: (a) its truth depends on the values of the variables x and y; (b) we need to formalize the meaning of the operators + and − as functions that map a pair of numbers to a number; (c) relational operators like > must be formalized as mapping pairs of numbers into truth values. The system of logic that can be interpreted by values, functions and relations is called first-order logic (also called predicate logic or the predicate calculus).

The predicate calculus includes a wider range of entities. It permits the description of relations and the use of variables. It also requires an understanding of quantification.

Predicate logic builds heavily upon the ideas of proposition logic to provide a more powerful system for expression and reasoning. A **predicate** is just a function with a range of two values, say false and true. We can use predicates in programming, e.g. in conditional statements of the form: if( p(...args ...)).

Here we are using the two possibilities for the return value of p, (true or false). We also use the propositional operators to combine predicates, such as in:

if( p(....) && ( !q(....) || r(....) ) )

**Predicate logic** deals with the combination of predicates using the propositional operators we have already studied. It also adds one more interesting element, the "quantifiers".

The meaning of predicate logic expressions is suggested by the following equation:

**Expression + Interpretation + Assignment = Truth Value**

An **interpretation** for a predicate logic expression consists of:

* a domain for each variable in the expression
* a predicate for each predicate symbol in the expression
* a function for each function symbol in the expression

Predicates are a type of function. They could be distinguished in predicate logic so as to separate predicates, which have truth values used by propositional operators, from functions that operate on arbitrary domains.

Note that the propositional operators are not counted as function symbols in the case of predicate logic, even though they represent functions. The reason for this is that we do not wish to subject them to interpretations other than the usual propositional interpretation.

**Predicate logic is sometimes called Predicate calculus that** allows us to reason about properties of objects and relationships between objects. In propositional calculus, we could express the English statement “I like cheese” by *A*. This enables us to create constructs such as *￢A*, which means “I do not like cheese,” but it does not allow us to extract any information about the cheese, or me, or other things that I like.

In predicate calculus, we use **predicates** to express properties of objects. So the sentence “I like cheese” might be expressed as

L(me, cheese)

where *L* is a predicate that represents the idea of “liking.” Note that as well as expressing a property of *me*, this statement also expresses a relationship between *me* and *cheese*.

The type of predicate calculus that we have been referring to is also called **first order predicate logic (FOPL)**.A first-order logic is one in which the quantifiers ∀ and  can be applied to objects or **terms**, but not to predicates or functions.

* 1. **Quantifiers**

There are a couple of assertions commonly made about a predicate: that it is **sometimes** true and that it is **always** true. For example, the predicate “x2 > 0” is **always** true when x is a real number. On the other hand, the predicate “2x -4 = 0” is only **sometimes** true; specifically, when x =2.

There are several ways to express the notions of “always true” and “sometimes true” in English.

 English Always True

For all n, P(n) is true. For all , x2 ≥ 0.

P(n) is true for every n. x2 ≥ 0 for every .

 Sometimes True

There exists an n such that P(n) is true. There exists an  such that 2x-4 =0.

P(n) is true for some n. 2x-4 =0 for some .

P(n) is true for at least one n. 2x-4 =0 for at least one .

All these sentences quantify how often the predicate is true. Specifically, an assertion that a predicate is always true, is called a universally quantified statement. An assertion that a predicate is sometimes true, is called an existentially quantified statement as described in section 4.2 below.

* 1. **Universal and Existential Quantifiers**

There are symbols or logical connectors to represent universal and existential quantification, just as there are symbols for “AND” (^), and so forth. In particular, to say that a predicate, P(x), is true for all values of x in some set, D, we write it as:



The universal quantifier symbol is read “for all,” so this whole expression is read “For all x in D, P(x) is true.” Remember that upside-down “A” stands for “All.”

Universal quantification makes statements about every object. Similarly, we can make a statement about *some* object in the universe without naming it, by using an existential quantifier. To say that a predicate P(x) is true for at least one value of x in D, we write: 

The existential quantifier symbol , is read “there exists.” So expression is read, as “There exists an x in D such that P(x) is true.” Remember that backward “E” stands for “Exists.”

The symbols  and  are always followed by a variable-typically with an indication of the set the variable ranges over-and then a predicate. For instance, “If you can solve any problem we come up with, then you get an A for the course.” can be written as follows:



Or may be



* 1. **Predicate Logic as Formal Language**
* Syntax
	+ Rules for constructing legal sentences in the logic
	+ ;2Which symbols we can use (English: letters, punctuation)
	+ How we are allowed to combine symbols
	+ How we interpret (read) sentences in the logic
	+ Assigns a meaning to each sentence,

 Example: “All lecturers are seven foot tall”

* + A valid sentence (syntax)
	+ And we can understand the meaning (semantics)
	+ This sentence happens to be false (there is a counterexample)

The syntax of predicate logic can be broken up into a vocabulary and a set of rules for constructing formulas out of that vocabulary. Basically, the primitives are constants, variables, predicates, connectives, quantifiers, and delimiters. Constants and variables correspond to objects in our world. Constants are like names, e.g. Ernie or Hortence. Variables are more like pronouns, e.g. they or it. Predicates allow us to describe properties of objects and sets of objects. Quantifiers allow us to refer to sets of things. The connectives and delimiters are the same from sentential logic.

These are the elements of the vocabulary of first-order predicate logic:

Constants a, b, c, . . .; with or without superscripts. The superscripts allow us to convert a finite set of letters into an infinite set of constants.

Variables x, y, z, . . .; with or without superscripts. Again, the superscripts allow us to convert a finite set of letters into an infinite set of variables.

* 1. **Proofs and Derivation Rules on Predicate wff and Verbal Arguments**

Let’s now look at some simple proofs using this new machinery. First, we consider an example of Universal Instantiation. We prove H(a) from G(a) and (∀x)(G(x) → H(x)).

Proofing:

1 (∀x)(G(x) → H(x)) Given

2 G(a) Given

3 G(a) → H(a) 1, U.I.

4 H(a) 2,3, M.P.

First, we remove the universal quantifier with Universal Instantiation and then use Modus Ponens to get the desired conclusion.

Next, we have an example of Universal Generalization. We try to prove

(∀x)(R(x) → W(x)) from (∀x)(R(x) → Q(x)) and (∀x)(Q(x) → W(x)).

Proofing:

1 (∀x)(R(x) → Q(x)) Given

2 (∀x)(Q(x) → W(x)) Given

3 (R(v) → Q(v)) 1, U.I.

4 (Q(v) → W(v)) 2, U.I.

5 (R(v) → W(v)) 3,4 H.S.

6 (∀x)(R(x) → W(x)) 5 U.G.

First, we use Universal Instantiation on the two initial assumptions, massaging them into a form appropriate for Hypothetical Syllogism. We then use Universal Generalization to convert that result back into a universally quantified expression. Notice how we judiciously chose to instantiate to v, anticipating that we would be using Universal Generalization later.

Finally, we consider a case of Existential Instantiation (E.I.). We prove

((∃x)P(x) ∧ (∃x)Q(x)) from (∃x)(P(x) ∧ Q(x)).

1 (∃x)(P(x) ∧ Q(x)) Given

2 (P(w) ∧ Q(w)) 1 E.I.

3 P(w) 2 Simp.

4 (∃x)P(x) 3 E.G.

5 Q(w) 2 Simp.

6 (∃x)Q(x) 5 E.G.

7 ((∃x)P(x) ∧ (∃x)Q(x)) 4,6 Conj.

First, we use Existential Instantiation to strip the quantifier and replace the variable with a new constant. We then split off the conjuncts with Simplification and use Existential Generalization on each to add separate new existential quantifiers. We then conjoin the results with Conjunction.

Notice that the basic strategy in most proofs is fairly clear. Simplify the initial formulas so that quantifiers can be removed. Manipulate the instantiated formulas using the Laws and Rules from the preceding chapter.

Finally, generalize to appropriate quantifiers.

 **Chapter 5**

 **Logic Programming**

* State the difference between procedural and declarative programming languages.
* How procedural programming languages solve problem?
* What is a recursive function?

The idea of logic programming is to use a computer for drawing conclusions from declarative descriptions. Such descriptions called logic programs consist of finite sets of logic formulas.

Logic programming is a technology that comes fairly close to embodying the declarative ideal that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge. The ideal is summed up in Robert Kowalski's equation,

*Algorithm* = *Logic + Control* .

* 1. **Procedural vs. Declarative Languages**

**Procedural Language** is a set of instructions that are executed one by one from beginning to end unless an instruction forces the control elsewhere.

Examples: Fortran, COBOL (old versions), Pascal, C, Perl, etc.

**Declarative Languages** use the principle of **logical reasoning** to answer queries. Logical reasoning is based on deduction.

For example: If (A is B) and (B is C), then (A is C).

* 1. **Programming Basics: PROLOG**

**Prolog** is the most widely used logic programming language. It is used primarily as a rapid-prototyping language and for symbol manipulation tasks such as writing compilers (Van Roy, 1990) and parsing natural language (Pereira and Warren, 1980). Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat different from standard predicate logic. Prolog uses uppercase letters for variables and lowercase for constants. Clauses are written with the head preceding the body; "**:-**" is used for left implication,commas separate literals in the body, and a period marks the end of a sentence.

PROLOG (PROgramming in LOGic) is the best known example of logic programming language. A logic program is a set of specifications in formal logic. Indeed the name itself comes from PROgramming LOGic. The intellectual roots of PROLOG reside in the theoretical concepts of using logic for problem specification.

PROLOG was developed in 1972 by **Alain Colmerauer**. The name was chosen by **Phillippe Roussel** as an abbreviation for PROgramming in Logic. It emerged from collaboration between Colmerauer & Robert Kowalski. Colmerauer was working on Natural Language Understanding using logic to represent semantics and using resolution for question-answering. Also, both developed the procedural interpretation of implications. This dual *declarative / procedural* interpretation later became formalized in the PROLOG which can be read and used both declaratively and procedurally.

The first PROLOG program, also written in 1972 and implemented was a *French- Questioning Answering System* as part of a project in natural language understanding. The major development of the PROLOG language was carried out from 1975 to 1979 at the department of AI of the University of Edinburgh.

PROLOG and other logic based languages supports a ***declarative programming style***- that is constructing a program in terms of high level description of a program’s constraints- rather than a procedural programming style-writing programs as a sequence of instructions for performing an algorithm.

This mode of programming essentially tells the computer “what is true” and “What needs to be done” rather than “how to do it”. This allows programmers to focus on problem solving as sets of specifications for a domain rather than the details of writing low level algorithmic instructions for “what to do next”

PROLOG is primarily an interpreted language. But some versions of PROLOG allow compilation of part or all of the set of specifications for faster execution. PROLOG is an interactive language; the user enters queries in response to the PROLOG prompt ?-. Prolog programs, state and query relations. The specifics of how the queries are answered is up to the implementation and its theorem-prover.

PROLOG is designed to enable programmers to build a database of **facts** and **rules**, and then to have the system answer questions by a process of logical deduction using the facts and rules in the database.

Facts entered into a PROLOG database might look as follows:

***tasty (cheese).***

***made\_from (cheese, milk).***

***contains (milk, calcium).***

These facts can be expressed as the following English statements:

***Cheese is tasty.***

***Cheese is made from milk.***

***Milk contains calcium.***

Facts are rules without a body: likes(max, julia)., likes(max, amabel).

They are always true, so we could also write them as: likes(max, julia) :- true. likes(max, amabel) :- true.

Facts with *arguments* are used to describe ***relationships***between arguments. e.g. lives\_in(max, london). likes(max, amabel). child(charles, amy, brian). price(template, 3, 4.75). assembly(arm, joint(ball,3)).

Facts are rules that are always true.

We can also specify rules in a similar way, which express relationships between objects and also provide the instructions that the PROLOG theorem prover will use to answer queries. The following is an example of a rule in PROLOG:

***contains (X, Y) :- made\_from (X, Z), contains (Z, Y).***

This rule is made up of two main parts, separated by the symbol “:-”. The rule thus takes the form:

***B :- A*** which means

***“if A is true, then B is true,” or “A implies B.”*** Hence, the rule given above can be translated as

***“If X is made from Z and Z contains Y then X contains Y.”***

Having entered the three facts and one rule given above, the user might want to ask the system a question: ***?- contains (cheese, calcium).***

Using a process known as **resolution** , the PROLOG system is able to use the rule and the facts to determine that because cheese is made from milk, and because milk contains calcium, therefore cheese does contain calcium. It thus responds: ***yes***

* 1. **Horn Clauses: Prolog Facts, Rules and Queries**

Prolog Programs are constructed from a number of ***clauses***: <head> :- <body>A program consists of clauses, and there are three forms: ***facts, rules*** and ***questions.***

* + ***hypotheses* (facts)**
	+ ***conditions* (rules)**
	+ ***goals: questions or queries****.*

Both <head*>* and <body> are composed of ***relationships***(also called *predications* or *literals*). Horn clauses, also called Prolog clauses, have at most one positive literal.

There are many ways to represent the predicates in a database, such as by structured files representing tables, spreadsheet subsections, etc. In the language **Prolog**, one of the ways to represent a predicate is just by **enumerating** all combinations of values for which the predicate is true.

A *predicate* is a collection of clauses with the same *functor* (name) and *arity* (number of arguments).

**The predicate Facts/ enumerating:**

**Facts** contain one atomic formula only, and they are expressed very like atomic formulas in predicate calculus. They start with predicate name, followed by argument list, enclosed in parentheses. As in logic, anything that can be an argument of a predicate is called as a *‘****term****’*.

E.g.: facts are used to convey information such as “lions are mammals”, “Miny likes mango”. In Prolog these facts are expressed as follows: *mammal(lion).likes(miny, mango);* In prolog all terms must be **atoms, numbers, variables, structures**.

Other examples:

 loves(john, mary).
 loves(mary, bill).
 loves(chuck, X) :- female(X), rich(X).

* **Atoms** are individuals and start with a *lowercase letter*. In the preceding example, john, mary, bill, and chuck are all atoms. Atoms are never capitalized. However everything enclosed in quotes is also considered atoms. For eg: capital(‘Ethiopia’, ‘Addis Ababa’) is acceptable;
* A **variable** starts with *uppercase letter* and indicates that the clause holds for every possible individual. For eg: likes(X, mango). Means everybody likes mango. **Every clause is universally quantified over all variables appearing in the clause.** This means that a variable can be instantiated to any term. Atoms are never capitalized, since they would make the atom a variable.
* Prolog uses **numbers**, both integers as well as floating point numbers. There are several predicates especially designed to deal with numbers. Numbers can be used in predicates like any other term.
* Prolog uses **structure**, to hold a collection of logically related data items.

For eg: book(title,author\_name,edition,price). Here *book* is a structure.

NB:

* Capitalization is *meaningful*! Variables begin with a capital letter:
 X, Socrates, \_result
* Atoms do *not* begin with a capital letter: x, socrates
* No space is allowed between a functor and its argument list:
 man(socrates), *not* man (socrates).
* Double quotes indicate a list of ASCII character values, *not* a string
* Don’t forget the period at the end of each horn clause.
* Program is facts + rules. (horn clauses).
* Clearly Prolog has something to do with logic Operators:
	+ Implication  **:-** to mean if
	+ Conjunction **,** for satisfaction
	+ Disjunction **;** to get other solutions.
* Prolog's "Yes" means "I can prove it"
* Prolog's "No" means "I can't prove it"

Let us define the predicates *mother* and *father* in this fashion. These predicates provide a way of modeling the family "tree" on the bottom.

mother(alice, tom).

mother(alice, carol).

mother(carol, george).

mother(carol, heather).

mother(susan, hank).

father(john, tom).

father(john, carol).

father(fred, george).

father(fred, heather).

father(george, hank).

****

**Figure 1: A family "tree" modeled as two predicates, mother and father.**

It is possible for a query to contain no variables, in which case we would expect an answer of 1 or 0. For example,

mother(susan, hank) ⇒ true

mother(susan, tom) ⇒ false

More interestingly, when we put variables in the queries, we expect to get values for those variables that satisfy the predicate:

mother(alice, X) ⇒ X = tom; X = carol (two *alternatives* for X)

father(tom, X) ⇒ false (no such X exists)

mother(X, Y) ⇒ (several *alternative combinations* for X, Y)

X = alice, Y = tom;

X = alice, Y = carol;

X = carol, Y = george;

X = carol, Y = heather;

X = susan, Y = hank

Note that the X and Y values must be in correspondence. It would not do to simply provide the set of X and the set of Y separately.

The Prolog language allows us to present queries and have answered them automatically in a style similar to the above.

The **Prolog Rules:**

**Prolog uses Rules**. Rules are really conditionals. Rules are usually expressed in the form of IF . . . THEN . . . statements with **implication** operator, such as: IF A THEN B. This can be considered to have a similar logical meaning as the following: **A→B**. A is called the *antecedent* and B is the *consequent* in this statement.

Rules consist of a head, which expresses the consequent of the conditional and a body which forms the antecedent.

For e.g. if rosy eats chocolate, then she gains weight,can be expressed as

 gainsweight(rosy):-eats(rosy,chocolate).

* + The head part of the above rule is – gainsweight(rosy), and the body is eats(rosy,chocolate).
* The head of the rule is always a single atomic formula. The body of a rule is frequently a conjunction.

For eg : aunt(X,Y):-sister(X,Z),parent(Z,Y). This means that X is an aunt of Y if X is a sister of Z, and Z is a parent of Y.

* The atomic formulas appearing in the body of a rule are often called ***goals***.

In the above e.g. sister(X, Z) and parent (Z, Y) are the goals.

* Rules can also contain disjunctions.

For e.g. one can define parent(X, Y) by the rule

parent(X,Y) :- mother(X,Y);father(X,Y).

* + Instead of semicolon, different clauses could be used.

i.e.: Parent(X, Y):- mother(X, Y).

 Parent(X, Y):- father(X, Y).

* A list of clauses together describes a predicate is also called a procedure.

In expressing rules, the consequent usually takes the form of an **action** or a **conclusion**. In other words, the purpose of a rule is usually to tell a system (such as an **expert system**) what to do in certain circumstances, or what conclusions to draw from a set of inputs about the current situation.

In general, a rule can have more than one antecedent, usually combined either by AND or by OR (logically the same as the operators ^ and ˅). Similarly, a rule may have more than one consequent, which usually suggests that there are multiple actions to be taken.

Antecedent of a rule compares an **object** with a possible **value**, using an **operator**. For example, suitable antecedents in a rule might be

* + - IF *x* > 3
		- IF name is “Bob”
		- IF weather is cold
* Here, the objects being considered are *x*, name, and weather; the operators are “>” and “is”, and the values are 3, “Bob,” and cold.

An object is not necessarily an object in the real-world sense—the weather is not a real world object, but rather a state or condition of the world. An object in this sense is simply a variable that represents some physical object or state in the real world. An example of a **recommendation** rule, which takes a set of inputs and gives advice as a result:

* + - * IF name is “Bob”
			* AND weather is cold
			* THEN tell Bob ‘Wear a coat’.

The conclusion of the rule is actually an action, and the action takes the form of a recommendation to Bob that he should wear a coat.

* Rules can also be used to represent relations such as:
	+ - IF temperature is below 0
		- THEN weather is cold

A predicate rules is also defined by the following logical expression:

grandmother(X, Y) :- mother(X, Z), parent(Z, Y).

Here : - is read as "if" and the comma separating *mother* and *parent* is read as "and". This

says, in effect, "X is the grandmother of Y if X is the mother of (some) Z and Z is the

parent of Y". We have yet to define *parent*, but let's do this now:

parent(X, Y) :- mother(X, Y).

parent(X, Y) :- father(X, Y).

Here we have two separate logical assertions, one saying that "X is the parent of Y if X is the mother of Y", and the other saying a similar thing for father. These assertions are not contradictory, for the connective :- is "if", not "if and only if". However, the collection of all assertions with a given predicate symbol on the *lhs* exhausts the possibilities for that predicate. Thus, the two rules above together could be taken as equivalent to:

parent(X, Y) iff (mother(X, Y) or father(X, Y))

Given these definitions in addition to the database, we now have a "knowledge base" since we have rules as well as enumerations. We can query the defined predicates in the same way we queried the enumerated ones. For example:

grandmother(alice, Y) ⇒ Y = george; Y = heather

grandmother(X, Y) ⇒ X = alice, Y = george;

X = alice, Y = heather

X = carol, Y = hank

grandmother(susan, Y) ⇒ false

* Rule systems can work using forward chaining, backward chaining, or both. Forward chaining works from a set of initial facts, and works toward a conclusion. Backward chaining starts with a hypothesis and tries to prove it using the facts and rules that are available.

**Prolog Queries**:

**Queries** are used to derive conclusions from the database. A query can be treated as, a rule without head. It can contain several goals. Goals must be separated either by a comma or semicolon; depending on whether the query is a conjunction or disjunction. Queries are applied in query mode or interactive mode. Here the prompt is as ?-

Clauses summary:

3 clauses in prolog:-

* Fact, rule and query.
* These 3 clauses are connected.
* Fact- can be thought of as a rule without a body.
* Query- can be thought of as a rule without a head.
* Syntactical rules for queries and bodies of rules are identical.
* All clauses must terminate with a period (**.**)(full stop).
* Use Comment: for increasing clarity.
* In prolog comment starts with the symbol **%** and terminates with the end of the line.
* E.g. connecting facts, rules and query.

|  |  |  |  |
| --- | --- | --- | --- |
| **Facts** | **Rule** | **Query** | **System reply** |
| **human(socrates).****human(plato).** | **mortal(X):-human(X).** | **mortal(plato).** | **Yes.** |
| **mortal(Y).** | **Y=socrates;****Y=plato** |

**Resolution and Recursion**

**Resolution**

 Generally two clauses ‘a’ and ‘b’ can be *resolved* provided ‘a’ contains a goal that matches ‘b’. In this case, after proper instantiation, one can, replace the goal of ‘a’ being matched by the body ‘b’. The resulting clause is called the resolvent of ‘a’ with ‘b’.

Resolution is an inference rule that is both SOUND and COMPLETE.

SOUND = only true facts (logical consequences) are inferred

COMPLETE = ALL facts that follow CAN be inferred

Basically, resolution is a CANCELLING method:

Given: A OR B

 NOT A OR C

 ---------------

infer: B OR C

A and NOT A "cancelled" each other.

**Rules Resolved With Rule**

For e.g: consider the following database

buys(X,Y) :- opportunity(Y),likes(X,Y).-----------(1)

opportunity(Z) :-lowprice(Z),highquality(Z)--------(2)

opportunity(Z) :-recommended(Z).--------------------(3)

1. **To resolve the first clause with the second clause**

*First*: Goal that matches First and Second clauses *opportunity(Y).*

*Second*: To complete the match, this goal opportunity(Y) is unified with head, opportunity(Z).

 ‘Z’ is instantiated with ‘Y’ which yields opportunity(Y):- lowprice(Y),highquality(Y).

*Third*: Body of the Ist clause, replaces the goal opportunity(Y), which leads to

buys(X,Y):-lowwprice(Y),highquality(Y),likes(X,Y).

1. **To resolve the 1st clause with the 3rd clause.**

buys(X,Y) ;- recommended(Y),likes(X,Y).

**Rule Resolved With a Fact**

 Since facts have no bodies, the goal to be matched by the rule essentially disappears.

* Find the resolvant of the following two clauses:
	+ buys(Z,U):-forsale(U),likes(Z,U),good(U)
	+ forsale(hat).
	+ Goal forsale(U) is unified with forsale(hat) by the instantiation of U=hat. All instances of the variable are replaced by hat, and the goal forsale(hat) is dropped. The result is, buys(Z,hat):-likes(Z,hat),good(hat).

***Illustration***

Consider the database:

forsale(dress).

forsale(hat).

forsale(shoes).

likes(jim,shoes).

likes(mary,dress).

likes(mary,hat).

good(hat).

buys(X,Y):-forsale(Y),likes(X,Y),good(Y).

To derive the query **forsale(hat)** using resolution.

1. Goal forsale(Y) is unified with forsale(hat) by the instantiation of Y=hat. The result of this unification is **buys(X,hat) :- likes(X,hat),good(hat)**.[using resolution-the goal disappeared]
2. Next we have to resolve likes(X,hat)
3. likes(X,hat) is unified with likes(mary,hat) by the instantiation of X=mary. The result is **buys(mary,hat):-good(hat).**
4. Next to rsolve the goal good(hat).
5. good(hat) – goal being a fact, succeeds and the above clause is resolved ,which removes the goal from the clause in question. The body of the goal therefore becomes empty; and a clause with an empty body merely becomes a fact.

 That is **buys(mary,hat).**

1. This fact agrees with the query and the query succeeds.

The query succeeds if all goals can be resolved. Once all goals have been resolved, nothing remains. That is empty clause has been derived. Hence a query succeeds if it leads to the empty clause. Otherwise the query fails.

**Resolution Example:**

Anyone passing his history exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but John is lucky. Anyone who is lucky wins the lottery. Is John happy?

**1st step: Convert to predicate logic**

1. Anyone passing his history exams and winning the lottery is happy.

x Pass(x, History)Win(x, Lottery)Happy(x)

2. But anyone who studies or is lucky can pass all his exams.

x y Study(x)Lucky(x)Pass(x,y)

3. John did not study, but John is lucky

Study(John) Lucky(John)

4. Anyone who is lucky wins the lottery.

x Lucky(x)Win(x, Lottery)

**2nd step: Convert to CNF**

Eliminate implications:

1. x (Pass(x, History) Win(x, Lottery))Happy(x)

2. x y (Study(x)Lucky(x) Pass(x,y)

3. Study(John) Lucky(John)

4. x Lucky(x)Win(x, Lottery)

Move inward:

1. x Pass(x, History)Win(x, Lottery))Happy(x)

2. x y (Study(x) Lucky(x) Pass(x,y)

3. Study(John) Lucky(John)

4. x Lucky(x)Win(x, Lottery)

Standardize variables: no action needed

Move quantifiers left: no action needed except drop quantifiers

Skolemize: no action needed

Distribute over

1. Pass(x, History) Win(x, Lottery))Happy(x)

2. (Study(x) Pass(x,y)) (Lucky(x)Pass(x,y))

3. Study(John) Lucky(John)

4. Lucky(x)Win(x, Lottery)

Flatten nested conjunctions and disjunctions no action necessary

State as a set of disjunction of literals

1. Pass(x, History)Win(x, LotteryHappy(x)

2. a. Study(x)Pass(x,y)

2. b. Lucky(x)Pass(x,y)

3. a. Study(John)

b. Lucky(John)

4. Lucky(xWin(x, Lottery)

Standardize variables apart

1. Pass(x1, History)Win(x1, Lottery)Happy(x1)

2. a. Study(x2)Pass(x2,y1)

2. b. Lucky(x3)Pass(x3,y2)

3. a. Study(John)

b. Lucky(John)

4. Lucky(x4)Win(x4, Lottery)

Now in conjunctive normal form (CNF)

**3rd step: Resolution Proof Procedure**

Assert negation of goal

– In this case the goal is to prove

Happy(John)

– Add the clause

Happy(John) to the KB(knowledge base).

• Resolve clauses together until FALSE is Derived (using Resolution Proof Tree).

**Recursion:**

Recursion in any language is the ability for a unit of code to call itself, repeatedly, if necessary. Recursion is often a very powerful and convenient way of representing certain programming constructs.

In Prolog, recursion occurs when a predicate contains a goal that refers to itself.

As we have seen in earlier chapters, every time a rule is called, Prolog uses the body of the rule to create a new query with new variables. Since the query is a new copy each time, it makes no difference whether a rule calls another rule or itself.

* A recursive rule is one which refers to itself. That is rules in which the predicate to be defined appears on both sides of the : - is called recursive rules.
* Any procedure that contains recursive rules is called a **recursive procedure**. Recursive procedures contain both recursive clauses and non recursive clauses. The non recursive clauses form the basis of recursion and they are referred to as **base clauses.**
* **Loops can be replaced by recursion**. Prolog uses recursion where procedural languages would use loops which mean that recursion is fundamental for writing programs in prolog. To do recursion effectively prolog allows one to define recursive structures.
* One of the very basic recursive structures is the **list**. In prolog, list takes the place of arrays as used in procedural languages.

Example:

* A thing, T1, is contained in another thing, T2, if T1 is directly located in T2. (This is the boundary condition.)
* A thing, T1, is contained in another thing, T2, if some intermediate thing, X, is located in T2 and T1 is contained in X. (This is where we simplify and recurse.)

We will now express this in Prolog. The first rule translates into Prolog in a straightforward manner.

is\_contained\_in(T1,T2) :- location(T1,T2).

The recursive rule is also straightforward. Notice that it refers to itself.

is\_contained\_in(T1,T2) :- location(X,T2), is\_contained\_in(T1,X).

**LOOPS**

* Loops are seldom used in Prolog; even though there are some built in predicates that have the effect of a loop. Most prolog programmers use recursion in place of loops, since all loops can be converted to recursive function calls.

*Exercise 1:* Using recursion find the factorial of a number.

**factorial(0,1).**

**factorial(N,F) :-**

 **N>0,**

 **N1 is N-1,**

 **factorial(N1,F1),**

 **F is N \* F1.**

 The predicate *factorial* is defined in the file *fact-recursion.pl*

*Output:*

**1 ?- [fact-recursion].**

% fact-recursion compiled 0.00 sec, 820 bytes

Yes

**2 ?- factorial(6,Answer).**

Answer = 720 ;

*Exercise 2:* Using recursion find the nth number in the Fibonacci series.

fib(0,1).

fib(1,1).

fib(N,Result):-

 N>1,

 Nminus2 is N-2,

 Nminus1 is N-1,

 fib(Nminus2,FibNminus2),

 fib(Nminus1,FibNminus1),

 Result is FibNminus1 + FibNminus2.

The predicate *fib* is defined in the file *fibonacci-recursion.pl*

*Output:*

**% fibonacci-recursion compiled 0.00 sec, 0 bytes**

Yes

**49 ?- fib(5,Fibonacci\_number\_5th).**

Fibonacci\_number\_5th = 8

**Expert systems**

* Expert systems originated in the 1960s.
* Fields in which they are used include chemistry, geology, medicine, banking and investments, and insurance
* A type of computer application program that makes decisions or solves problems in a particular field, such as finance or medicine, by using **knowledge and analytical rules** defined by experts in the field. Human experts solve problems by using a combination of **factual knowledge and reasoning ability.** In an expert system, these two essentials are contained in two separate but related components, a *knowledge base* and an *inference engine.*
* The **knowledge base** provides specific facts and rules about the subject.
* The **inference engine** provides the reasoning ability that enables the expert system to form conclusions.
* Expert systems also provide additional tools in the form of user interfaces and explanation facilities.
	+ **User interfaces**, as with any application, enable people to form queries, provide information, and otherwise interact with the system.
	+ **Explanation facilities**, an intriguing part of expert systems, enable the systems to explain or justify their conclusions, and they also enable developers to check on the operation of the systems themselves.
* For example, an expert system that diagnoses blood disease in a patient would require a *knowledge base* that included data on physiology, blood pathogens, disease symptoms, and treatment options. The *inference engine* searches through the knowledge base and concludes which possible disease or diseases the patient has and then suggests various treatments based on that diagnosis.

 **Chapter 6**

 **Introduction to Other Logics**

Different logics exist, which allow you to represent different kinds of things, and which allow more or less efficient inference. These are: Propositional logic, Predicate logic, Temporal logic, Modal logic, Description logic.

* 1. **Modal Logic and its Syntax**

There are many forms of modal logic. Each one is based upon two parameters:

\_ W is the set of possible worlds (machine states, future times ...)

\_ R is the accessibility relation between worlds (state transitions, flow of time,...)

The pair (W; R) is called a modal frame.

The two modal operators, or modalities, are and .

\_ A means A is necessarily true

\_ A means A is possibly true

Here ‘necessarily true’ means ‘true in all worlds accessible from the present one’.

The modalities are related by the law ; in words, ‘it is not possible

that A is true’ is equivalent to ‘A is necessarily false.’

Complex modalities are made up of strings of the modal operators, such as

A , A, A, etc. Typically many of these are equivalent to other;

an important modal logic, A is equivalent to A.

Modal logics are used to formalize statements where finer distinctions need to be made than just ‘true’ or ‘false’. Classically, modal logic distinguished between statements that are necessarily true and those that are possibly true. For example, 1 + 1 = 2, as a statement about the natural numbers, is necessarily true because of the way the concepts are defined. But any historical statement like ‘Ethiopian lost the battle of poverty’ is only possibly true; if circumstances had been different, the outcome of poverty’ might have been different. Modal logics have turned out to be extremely useful in computer science. It is better to study a form of modal logic called temporal logic, where ‘necessarily’ is interpreted as always and ‘possibly’ is interpreted as eventually.

Modal logics are an extension of classical logic that allows us to reason about possibilities and certainties. In other words, using a modal logic, we can express ideas such as “although the sky is usually blue, it isn’t always” (for example, at night). In this way, we can reason about possible worlds. A possible world is a universe or scenario that could logically come about.

The following statements may not be true in our world, but they are possible, in the sense that they are not illogical, and could be true in a possible world:

Dogs can fly.

People have no legs.

It is possible that some of these statements will become true in the future, or even that they were true in the past. It is also possible to imagine an alternative universe in which these statements are true now.

* 1. **Fuzzy Logic**

Fuzzy logic can be conceptualized as a generalization of classical logic. This means that **Fuzzy logic** is a form of many-valued **logic** that deals with approximate, rather than fixed and exact reasoning. Compared to traditional binary **logic** (where variables may take on true or false values), **fuzzy logic** variables may have a truth value that ranges in degree between 0 and 1. On the other hand, FL could have the truth value of true (1), false (0) or both/uncertainty (1<x>0). Each fuzzy variable can take a value from 0 (not at all true) to 1 (entirely true) but can also take on real values in between. Hence, 0.5 might indicate “somewhat true,” or “about as true as it is false.”

**Fuzzy logic** may be viewed as an extension to classical logic systems, providing an effective conceptual framework for dealing with the problem of knowledge representation in an environment of uncertainty and imprecision. FL, as its name suggests, is a form of logic whose underlying modes of reasoning are approximate rather than exact. Its importance arises from the fact that most modes of human reasoning —and especially commonsense reasoning— are approximate in nature.

**Fuzzy logic** is an approach to a formal treatment of uncertainty, relies on quantifying and reasoning through natural language and uses linguistic variables to describe concepts with vague values. Such as: tall, large, small, heavy, ...

Fuzzy logic is a form of logic that applies to fuzzy variables. Fuzzy logic is nonmonotonic, in the sense that if a new fuzzy fact is added to a database, this fact may contradict conclusions that were previously derived from the database. On the other hand, nonmonotonicity deals with the inconsistencies in the knowledge base may arise as new sentences are added and sometimes remedied by truth maintenance systems.

* 1. **Intuitionistic Logic**

**Intuitionistic logic**, sometimes more generally called constructive **logic**, is a system of symbolic **logic** that differs from classical **logic** by replacing the traditional concept of truth with the concept of constructive provability.

Therefore, Intuitionistic logic is a weakening of classical logic by omitting, most prominently, the principle of excluded middle and the reduction ad absurdum rule. As a consequence, this logic has a wider range of semantical interpretations. The syntax of Intuitionistic logic is similar to that of the classical logic.

* 1. Lukasiewicz Logic

In which areas Lukasiewicz Logic could be applied? This logic could be applied in different types of logic. Lukasiewicz was American logician born in Bialystok, Poland who developed first **ternary predicate calculus** in 1920 many fundamental works on ***multiple-valued*** logic

* 1. **Probabilistic Logic (Bayesian Learning)**

Logic and probability theory are the main tools in the formal study of reasoning, and have been fruitfully applied in areas as diverse as philosophy, artificial intelligence, cognitive science and mathematics as we have learnt in chapter one.

The probabilistic treatment of uncertainty plays an important role in many applications of knowledge representation and reasoning. Often, we need to reason with uncertain information under partial knowledge and then the use of precise probabilistic assessments seems unrealistic. Moreover, the family of uncertain quantities at hand has often no particular algebraic structure. *Bayes’ theorem on probability*:



The ontological and epistemological commitments of five different logics are summarized below.

