BCS/CSC 227: LOGIC PROGRAMMING

Prepared by Dr. D. K. Muyobo

TOPIC 1: INTRODUCTION TO LOGIC PROGRAMMING

What is Logic Programming?

- Logic programming is a paradigm/particular way to approach programming
- The mindset of various paradigms is quite different and determines how we design and reason about programs.
  - Other paradigms we might compare to logic programming: imperative programming and/or functional programming

(1) **Imperative programming**

Most conventional programming languages (e.g. C, C++, Java)

Semantics: state based

Computation viewed as state transition process

Categories: Procedural, Object Oriented and Other non-structured

For example: C, Pascal, Turing are in the Procedural category, steps of computation describe state changing process

(2) **Declarative Programming**

Focus is on logic (WHAT) rather than control (HOW)

Categories:

Logic Programming: Computation is a reasoning process, e.g. Prolog

Functional Programming: Computation is the evaluation of a function, e.g. Lisp, Scheme, (C#: supports both imperative and functional programming approaches)

Constrained Languages: Computation is viewed as constraint satisfaction problem, e.g. Prolog (R)

Level of language

Low level - has a world view close to that of the computer

High level - has a world view closer to that of the specification (describing the problem to be solved, or the structure of the system to be presented)

Aspects of Logic Programming

- Programs are written in the language of some logic / A class of programming languages, and a subclass of the declarative languages, that is based on the use of logical formulas
- Execution of a logic program is a theorem proving process; that is, computation is done by logic inferences
- The interpreter is usually some version of resolution, or another logical inference process

   It is based on first order predicate logic

Why logic programming (LP)?

- A logic program is a specification of a solution to a problem, in addition, it is an executable specification
- Like Lisp, LP is about manipulation of symbols, and thus has potential in AI applications. Unlike Lisp, computations in LP are reasoning processes LP are important because of their:
  - declarative nature,
  - their potential power and flexibility,
  - and their suitability for execution on highly parallel architectures

Why Logic Programming is not as Popular….as C/Java

- Mistaken at first as some universal computer language
- Not yet as efficient as C
- Support to Prolog takes effort, resources; companies are not willing to pay for it Its value not recognized by industry.

Why is LP difficult, especially for beginners?

- The way to write logic programs departures dramatically from the traditional way of programming.
- The execution of logic programs is a reasoning process, resulting in nondeterministic computations.

History of Logic Programming

- Formulated in 1974 by a professor at Univ. of Edinburgh.
- First system implemented in 1995 by a research group in France. First compiler built in 1997 by a PhD student also in Edinburgh.
- Japan's fifth generation computer project announced in 1980.
- Efficiency improved in recent years
- Interfaces with other languages such as C/Java.

Task 1: What are some examples of logic programming languages?

TOPIC 2: UNDERSTANDING LOGIC PROGRAMMING

What is logic?

- **Logic** is a language. It has syntax and semantics. But more than a language, it has inference rules.

- **Syntax:** defines the rules about what are well-formed formulas; this is usually the easy part of a logic.
- **Semantics**: about the meaning carried by the formulas, mainly in terms of logical consequences.
- There are different ways to describe meanings, the most traditional one is called "**model theoretic**", and this is usually a difficult concept for newcomers.
- **Inference rules**: describe correct ways to derive conclusions - Given a collection of formulas, an inference rule allows one to derive new formulas
- Main focus in any **logic** is the **inference rules** that are **faithful** to the underlying **semantics**.

NB: *a set of inference rules must be sound with respect to the semantics. It's desirable if it is also complete in that any logical consequence can be derived*

History of logic

Logic has been studied since the classical period (600-300 BC)

The Greeks (Thales) were the first to formally analyze the reasoning process.

Aristotle (384-322 BC) the "**Father of Logic**" and many other Greeks searched for Universal truth that were irrefutable.

A second great period for Logic came with the use of symbols to simplify complicated logical arguments.

Gottfried Leibniz (1646-1716) began this work at age of 14, but failed to provide a workable foundation for **Symbolic Logic.**

George Boole (1815-1864) - considered the "**Father of Symbolic Logic**":

Why?

- He developed logic as an abstract mathematical system consisting of defined terms (propositions), operations (conjunction, disjunction, implication, negation etc.) and rules for using these operations.
- His basic ideal was that if simple propositions could be represented by precise symbols, then relation between the propositions could be read as precisely as an algebraic equation.
- Boole developed an "**Algebra of Logic**" in which certain types of reasoning were reduced to manipulations of symbols.

Symbolic Logic

Provides an idealized model of mathematical language and proof.

It is based on the on the theory of Syllogism. Which say:

Every man is an animal.
Every animal dies.
Therefore every man dies.

In symbolic Logic, we can introduce Variables A, B, C, . . . to stand for fundamental statements or propositions and Symbols ∧, ∨, ¬, and → to stand for "**and, or, not and if-then"** respectively.

General Pattern

Every A is B.
Every B is C.
∴ every A is C.
Represented as:

$$\frac{A \qquad\qquad B}{C}$$

Which indicates that statement C is a logical consequences of A and B.

Definitions

Argument	A series of Statement
Premises	All statements in an argument except the final one
∴	Symbol for "**therefore**": used to identify conclusion.
Modus Ponens	Latin for "method of affirming" a rule of inference used to draw logical conclusion, which states that's if p is true and if p→q then q is true as well.
Modus Tollens	Latin for "method of denying" a rule of inference used to draw logical conclusion from the combination of Modus Ponens and contrapositive. It states that's if **q** is **false** and if **p→q** then **p** is false as well.
Fallacy	An error in reasoning
Contradiction Rule	Given a statement p. if ¬p leads logically to a contradiction, then p must be true.

Testing for Arguments Validity

1. Identify the premises and conclusion of the arguments
2. Create a truth table showing the value of the premises and conclusion **3.** Locate the rows in which the premises are all true (the critical rows)
4. For each critical row, determines if the conclusion is also true.

Examples 1:

If I study hardy, then I will get an A	: p→q
I will study hard	p
Therefore, I will get an A.	∴ q

(Modus Ponens)

Example 2:

If is a car, then it has wheels, I does not have wheels. Therefore it is not necessarily a car.

(Modus Tollens – correct)

Common Fallacies

- Vague or un-ambiguous premises
- Using the opinion of an authority figure as evidence in support of an argument, even if the authority is not an expert in the relevant field.
- Restating the argument rather than providing evidence or support.
- Arguing that a claim is true because it has not been proven false, or vice versa.
- Jumping to conclusion: Drawing a conclusion based on a small sample size rather than a reliable, representative sample.
- Assuming the truth of the conclusion in the premise, often by restating the same idea in different words.
- Arguing that something is true or good because many people believe it.

Computation vs. Deduction

Examining the difference between computation and deduction helps understand logic programming.

Computation	**Deduction**
Means to compute: the process of executing or carrying out a set of instructions to perform a specific task.	Means to deduce: the process of drawing specific conclusions from general principles or statements.
To Compute, start from a given expression and, according to a fixed set of rules (the program) generate a result.	To deduce, start from a conjecture and according to a fixed set of rules (the axioms and inference rules), try to construct a proof of the conjecture
Computation is mechanical and requires no ingenuity	Deduction is a creative process

- **Focus:** Computation is concerned with the execution of instructions, while deduction is concerned with drawing logical inferences.
- **Execution vs. Inference:** Computation involves the practical execution of tasks, while deduction involves logical inference based on rules and facts.
- **Process:** Computation in logic programming involves the step-by-step execution of instructions, while deduction involves the logical reasoning process of deriving new information from existing knowledge.

Connection between Computation and Deduction

- Philosophers, mathematicians, and computer scientists have tried to unify the two/ to understand the relationship between them for centuries.
  - **Example** -George Boole2 succeeded in reducing a certain class of logical reasoning to computation in so-called Boolean algebras

    - Not everything we can reason about is in fact mechanically computable, even if we follow a well-defined set of formal rules.
- Computation and deduction are interconnected in logic-based systems, particularly in logic programming. Computation involves the practical execution of instructions, and deduction involves logical inference to derive new information. The effective use of logical reasoning during computation is what enables logic-based systems to perform tasks such as solving queries, making decisions, or deriving new facts.

Different connection:

- Computation can be seen as a limited form of deduction because it establishes theorems.

For example, 15 + 26 = 41 is both the result of a computation, and a theorem of arithmetic.

- Deduction can be considered a form of computation if we fix a strategy for proof search, removing the guesswork (and the possibility of employing ingenuity) from the deductive process.
- This latter idea is the foundation of logic programming.
    - Logic program computation proceeds by proof search according to a fixed strategy. By knowing what this strategy is, we can implement particular algorithms in logic, and execute the algorithms by proof search

Judgments and Proofs

- Since logic programming computation is proof search, to study logic programming means to study proofs.
- The most basic notion is that of a judgment, which an object of knowledge is. We know a judgment because we have evidence for it. The kind of evidence we are most interested in is a proof

Proof Search

- To make the transition from inference rules to logic programming a particular strategy need to be imposed.
- Proof search plays a critical role in automated reasoning systems, logic programming languages, and formal methods, enabling the automated discovery of logical proofs and solutions to complex problems. It involves navigating through the logical structure of formulas and rules to find a valid solution or demonstrate the validity of a statement.

Fundamental Approaches:

- Search backward from the conjecture, growing a (potential) proof tree upwards (Backward chaining/goal-directed/ top-down), or
- we could work forward from the axioms applying rules until we arrive at the conjecture (forward-reasoning (Chaining)/data-driven/bottom-up).
- Logic programming was conceived with goal-directed search - this is still the dominant direction since it underlies Prolog - the most popular logic programming language

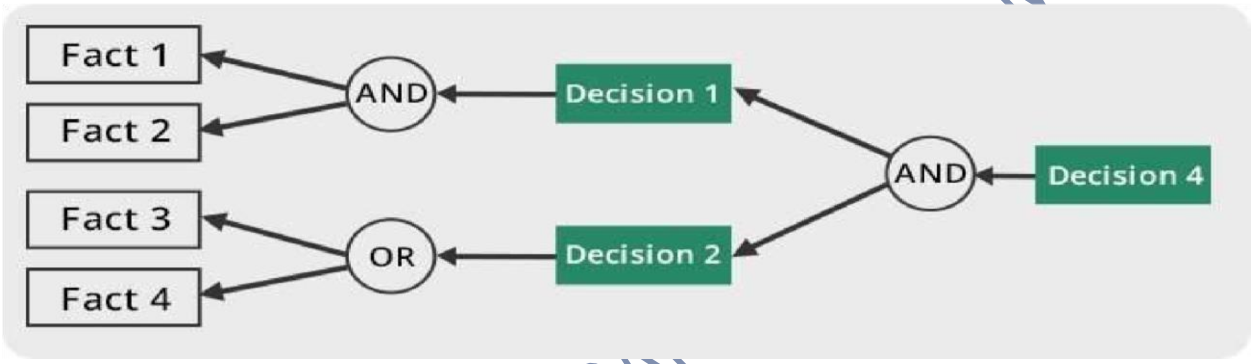
Review on Strategies used by the Inference Engine

Inference is accomplished by a process of chaining through rules recursively either in a forward or backward direction.

A Backward Chaining system (a goal driven system)

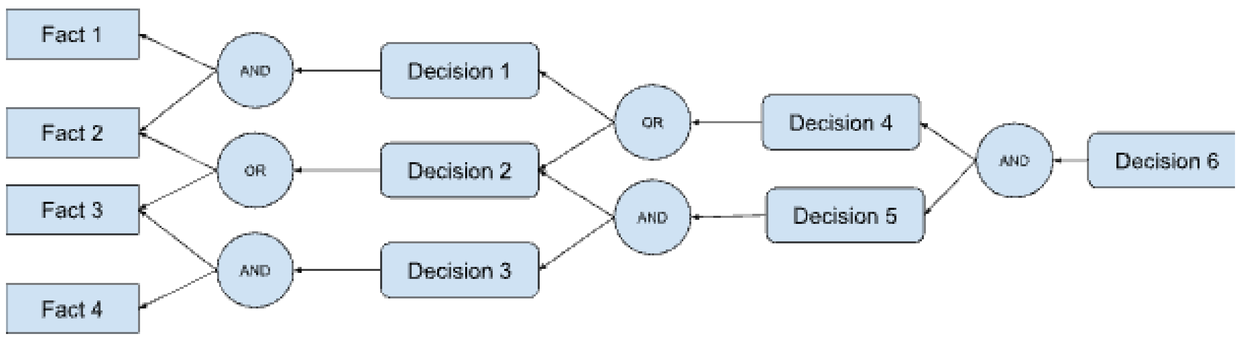
- With this strategy, an expert system finds out the answer to the question, "Why this happened?"
- Works with the system assuming a hypothesis of what the likely outcome will be, and the system then works backwards to collect the evidence that would support this conclusion.

   Expert systems used for planning often use backward chaining.



Forward Chaining expert system (a data driven system)

- It is a strategy of an expert system to answer the question, "What can happen next?"
- Simply gathers facts (like a detective at the scene of a crime) until enough evidence is collected that points to an outcome.
- It is often used in expert systems for diagnosis, advise and classification, although the size and complexity of the system can play a part in deciding which method of inferencing to use.



TOPIC 3: PROPOSITION LOGIC

Definition - A proposition is a declarative sentence that is either true (denoted either T or 1) or false (denoted either F or 0).

– Notation: Variables are used to represent propositions. The most common variables used are p, q, and r.

Examples of Propositions

1. The Earth is further from the sun than Venus
2. Do you want to go to the movies?
3. There is life on Mars
4. $2 \times 2 = 5$
5. Clean up your room
6. $x + 2 = 2x$ when $x = -2$
7. $2x = 2 + x$
8. This sentence is false

Propositions: Answer

- The Earth is further from the sun than Venus
- There is life on Mars
- $2 \times 2 = 5$
- $x + 2 = 2x$ when $x = -2$

Sentences that are not propositions.

- "Do you want to go to the movies?" Since a question is not a declarative sentence, it fails to be a proposition. • "Clean up your room." Likewise, an imperative is not a declarative sentence; hence, fails to be a proposition
- "$2x = 2 + x$." This is a declarative sentence, but unless x is assigned a value or is otherwise prescribed, the sentence neither true nor false, hence, not a proposition. • "This sentence is false." What happens if you assume this statement is true? false? This example is called a paradox and is not a proposition, because it is neither true nor false.
-

Alphabet

– Variables, e.g. p, q, r, ..., p1, ..., p', ...

– Constants: T and F or 1 and 0 – Connectives: {¬,∧,∨,→,≡}

- or {~, &, #, ->, <-> in some books}

– Brackets: (and)

Well-formed-formula (wff)

– All variables and constants are wffs.

– If A and B are wffs, then the following are also wffs.

- $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \equiv B)$

– Priority of connectives, and rules for removing brackets

Semantics and truth tables

- true (1) and false (0)
- state
- Tautologies: true in all possible states

Satisfiable

- A formula A is satisfiable iff there is at least one state v where v(A)=true
- A set of formulae X is satisfiable (or consistent) iff there is at least one state v where for every formula A in X, v (A)=true.

- Contradiction: (unsatisfiable, inconsistent)
    - If A is a tautology, ¬A is a contradiction

Why predicate over propositions

1. The propositional logic is not powerful enough to represent all types of assertions that are used in computer science and mathematics, or to express certain types of relationship between propositions such as equivalence
    - For example, **the assertion "x is greater than 1", where x is a variable, is not a proposition because you can not tell whether it is true or false unless you know the value of x. Thus the propositional logic can not deal with such sentences. However, such assertions appear quite often in mathematics and we want to do inferencing on those assertions.**
2. the pattern involved in the following logical equivalences cannot be captured by the propositional logic:

- "Not all birds fly" is equivalent to "Some birds don't fly".
- "Not all integers are even" is equivalent to "Some integers are not even".
- "Not all cars are expensive" is equivalent to "Some cars are not expensive", Each of those propositions is treated independently of the others in propositional logic. For example, if P represents "Not all birds fly" and Q represents "Some integers are not even", then there is no mechanism in propositional logic to find out tha P is equivalent to Q. Hence to be used in inferencing, each of these equivalences must be listed individually rather than dealing with a general formula that covers all these equivalences collectively and instantiating it as they become necessary, if only propositional logic is used.
- Thus we need more powerful logic to deal with these and other problems. The predicate logic is one of such logic and it addresses these issues among others.

TOPIC 4: PREDICATE LOGIC

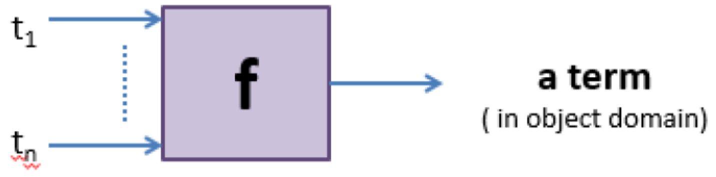
- a **predicate** is commonly understood to be a Boolean-valued function *P*: *X*→ {true, false}, called the predicate on *X*.
- However, predicates have many different uses and interpretations in mathematics and logic, and their precise definition, meaning and use will vary from theory to theory.
- Informally, a predicate is a statement that may be true or false depending on the values of its variables.
  - It can be thought of as an operator or function that returns a value that is either true or false.
  - For example, predicates are sometimes used to indicate set membership: - Thus, a predicate P(x) will be true or false, depending on whether x belongs to a set.
- Predicates are also commonly used to talk about the properties of objects, by defining the set of all objects that have some property in common.
  - example, when P is a predicate on X, one might sometimes say P is a property of X. Similarly, the notation P(x) is used to denote a sentence or statement P concerning the variable object x. The set defined by P(x) is written as {x | P(x)}, and is just a collection of all the objects for which P is true.
- For instance, {x | x is a natural number less than 4} is the set {1,2,3}.
  - If t is an element of the set {x | P(x)}, then the statement P(t) is true.
  - Here, P(x) is referred to as the predicate, and x the subject of the proposition (Sometimes, P(x) is also called a propositional function, as each choice of x produces a proposition)

Alphabet

- Alphabet of propositional logic
- Object variables, e.g. x, y, z, ..., x1, ..., x', ... – Object constants, e.g. a, b, c, ...
- Object equality symbol =
- Quantifier symbols ∀ (and ∃)
- and some functions & predicates

Term

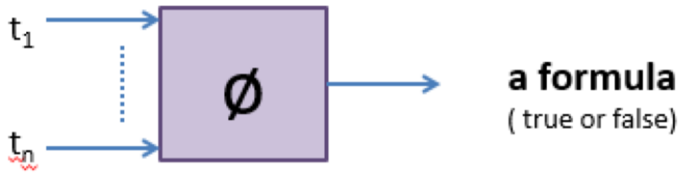
- An object variable or constant, e.g. x, a
- A function f of n arguments, where each argument is a term, e.g. f(t1, t2, ...tn)



Atomic formula

- A Boolean variable or constant

- The string t = s, where t and s are terms
- A predicate ø of n arguments where each argument is a term , e.g. ø(t1, t2, ...tn)



Well-formed formula

- Any atomic formula
- If A and B are wffs, then the following are also wffs. $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \equiv B)$, $((\forall x)A)$, $((\exists x)A)$

• Examples

Numbers

- Object constants: 1, 2, 3, ...
- Functions: +, -, *, /, ... – Predicates: >, <, ...
- Examples of wffs: $> (x, y) \rightarrow > (+(x,1), y)$

• Or the familiar notation: $x > y \rightarrow x + 1 > y$

• Another example: $y\ x! = z \rightarrow (x+1)! = (x+1)* z$

Sets

- Object constants: {1}, {2,3},...
- Functions: U, I,...
- Predicates: ⊂, ⊆,...
- A wff: $(x \text{ I } y) \subseteq (x \text{ U } y)$

• …More example

Our world

- Object variables: X, Y, ...

• upper case in PROLOG

- Constants such as: john, mary, book, fish, flowers, ...

• Note lower case in PROLOG

- Functions: distance(point1, X), wife(john)
- Predicates: owns(book, john), likes(mary, flowers), ...
- true and false in PROLOG
  - Relative to PROLOG’s knowledge of the world
  - False whenever it cannot find it in its database of facts (and rules)

DO IT YOURSELF [WARM UP EXERCISE]

a. What is symbolic logic? Give the general pattern used in representing the theory of syllogism.

b. Using appropriate notations, translate the following English sentence into a logical expression and draw its truth table "**You can access the Internet from campus** only **if you are a computer science major** or **you are not a freshman**."

c. Differentiate between Modus Ponens and Modus Tollens using relevant arguments.

d. Construct the truth table for the following two formulas **$(p \wedge \neg (q \vee r))$ and $(\neg p \vee (q \vee r))$.** Say for each one if it is a tautology, contingency, satisfiable, contradiction or logical consequence of the other.

e. [**The Kakamega Guardians]** you are walking around **Kakamega** and all of a sudden you find yourself in-front of three possible roads: The road on your left is paved with gold, the one in-front of you is paved with marble, while the one on your right is made of small stones. Each road is guarded by a guardian. You talk to the guardian and this is what they tell you:

The guardian of the gold road: "*This road will bring you straight to MMUST. Moreover, if the stones takes you to MMUST, then also the marble takes you to MMUST.*"
The guardian of the marble road: *"Neither the gold nor the stone roads will take you to MMUST."*
The guardian of the stone road: *"Follow the gold and you will reach MMUST, follow the marble and you will be lost."*

i Using the appropriate notations, formalize the guardians' responses and write down its conjunction.

ii Given that you know that all the guardians are liars, can you choose a road being sure that it will lead you to **MMUST**? If this is the case which road will you choose?

f. Three students **Andrew**, **Brian** and **Christian** are accused of introducing a virus in the Computer Forensics Lab. During the interrogation they make the following claims:

Andrew says: "*Brian did it and Christian is innocent*"
Brian says: "*If Andrew is guilty then Christian is guilty too*".
Christian says: "*I did not do it. One of the others or maybe both of them did it*"

i. Write a formula in propositional logic then represents the conjunction of the three above claims using the following atomic propositions: A: Andrew is guilty, B: Brian is guilty and C: Christian is guilty.

ii. Are the three above statements contradictory? Justify.

iii. Assuming that nobody lied, who is innocent and who is guilty? Justify

g. Differentiate between propositional and predicate logic and explain any two limitations of propositional logic that can be overcome by predicate logic.

h. To make the transition from inference rules to logic programming a particular strategy need to be imposed. Discuss the two fundamental strategies used by the Inference Engine.

i. What kinds of knowledge can be represented in propositional logic?

j. Explain how judgment and proofs are handled in logic programming.

k. Why is George Boole considered as the "**father of symbolic logic**"?

l. Express the following statements in predicate logic

  i "Everybody must take a discrete mathematics course or be a computer science student".

  ii There is someone loved by everyone

m. First Order Predicate Calculus is the basis of almost all knowledge representation and reasoning in every area of symbolic Artificial Intelligence (AI). Give at least four area of AI where this can be applied.

TOPIC 5: INTRODUCTION TO PROLOG

What is Prolog?

- **Prolog:** Programmation en Logique (Programming in Logic). It is a declarative programming language
- **In a declarative language**
  - the programmer specifies a goal to be achieved (programs specify **what** the computer should do (the *goal*), not **how** it should be done (the steps).
  - the Prolog system works out how to achieve it
- Goals are described by assertions (rules) that state the characteristics of the goal i.e. Declarative programming is sometimes called **rule-based programming**.
- The rules are formulated based on the predicate calculus which is a form /principles of symbolic logic.

Background of Prolog

- Developed in the 1970s
- Originated in the field of language processing
- Other important application areas: artificial intelligence, databases.
- AI programs, especially expert systems, are often structured as a knowledge base (collection of facts) and a set of rules of the form "if X then Y".
- Database programs also have a knowledge base of specific facts (the database) and a set of rules that express relations between entities.
- SQL (Structured Query Language): a declarative language used in database applications is another well-known example in the declarative paradigm. ○ Many of SQL's elements (clauses, predicates, queries) also are found in Prolog ○ SQL, however, is not Turing complete whereas Prolog is

Applications of Prolog

Some applications of Prolog are:

- intelligent data base retrieval
- natural language understanding
- expert systems
- specification language
- machine learning
- robot planning
- automated reasoning
- problem solving

Characteristics of Prolog

- A particular problem can be solved using prolog in a few lines of code.
- It is an important tool to develop AI applications and Expert Systems
- Prolog program consists of facts and rules to solve the problem and out is all possible answers to the problem.
- Prolog language is a declarative in nature, use the inference depending on facts and rules.
- Programs written in logic programming languages exhibit ○ Non-determinism: There may be several solutions, or acceptable goal states, based on use of different rules/facts
    - ○ Backtracking: the problem-solving mechanism built into Prolog

Data types in Prolog Prolog supports:

- **Integers** {1,2,3, . . .}
- **Real numbers** (decimal values)
- **Characters** both lower and upper case
- **Strings**
- **Symbols** (single or sequence of characters surrounded with quotes)

Logic Systems-Propositional Logic

- Propositional logic (PL) is a formal reasoning system based on *propositions:* assertions that are either true or false;
- Propositions are
    - – declarative statements; similar to sentences in natural languages
    - – composed of constants
- Propositions are combined using logical operators *and, or, if/then, not*
- PL is the basis for logic (Boolean-valued) expressions in traditional programming languages.
- **Examples of propositions:**

"Jane is a parent" = proposition p
"John is a parent" = proposition
q Combining propositions if (p && q) then . . .

In Prolog, propositions are written without variables:

```
parent(jane).   parent(john).
```

Propositions are either true or false but Prolog **facts** are **true propositions**.

```
man(jack).
```

Prolog queries can ask if a certain proposition is true or false

```
?- man(jack).
```

Logic Systems - Predicate Calculus

- Predicate calculus (PC), or first-order logic (FOL), extends propositional logic
- A predicate is a declarative statement with one or more variables, which has a truth value when the variables are instantiated
  - parent(X) (the predicate)
  - parent(jane) or parent(john) instantiate the predicate with values, permitting the truth value to be determined.
- Predicate calculus also includes quantifiers:
  - for all: ∀
  - there exists: ∃
- Quantifiers and predicates extend the expressive power: – ∀ x *if* Parent(x) *then* HasChild(x) – In Prolog, quantifiers are implied:
  - Parent(X) :- HasChild(X)

    This is equivalent to the previous logical statement
- PC/FOL is powerful enough to describe most mathematical domains Examples:
- ∀x (speaks(x, Russian)) ∀ is the universal quantifier
- ∃x (speaks (x, Russian)) ∃ is the existential quantifier.
- ∀x (¬literate(x) ⊃

  (¬writes(x) ∧ ¬ ∃y(reads(x,y) ∧ book(y))))
- The truth of these propositions depends on the values of x and y.

Logic and Horn Clauses

- Prolog syntax is based in large part on a variant of predicate logic known as the Horn clause.
- **Definition:** A Horn clause has a head *h*, which is a predicate, and a body, which is a list of predicates $p_1, p_2, \ldots, p_n$.
- Written as $h \leftarrow p_1, p_2, \ldots, p_n$
- Meaning: *h* is true if all the *p*'s are true
- Example: *snowing(C) ← freezing(C), precipitation(C)*
- Horn clauses are restricted to have 0 or one predicate on the left hand side.
  - **0**: headless Horn clauses:

    dog(Spot)
  - **1**: headed Horn clauses correspond to rules – right hand side is a conjunction of predicates. In Prolog, we would write

```
human(X) :- man(X)
snowing(C):- freezing(C), precipitation(C)
```

Resolution and Unification

- Logic based systems provide a formal notation for expressing propositions (facts) and predicates (rules), as well as a formal method for making inferences based on the facts and rules.
- **Resolution** is the process of making an inference from two Horn clauses

Resolution, Instantiation, and Unification

- Definition: when applied to Horn clauses, resolution says that if *h* is the head of one Horn clause and it matches a term in another Horn clause, then that term can be replaced by (the RHS of) *h*
- In general, from the Horn clauses
  - $h \leftarrow p_1 \ldots p_2$ and $t \leftarrow t_1, h, t_2$ we can infer $t \leftarrow t_1, p_1 \ldots p_2, t_2$

Resolution Example

- Given the following propositions:

  older(x, y) ← parent(x, y)
  wiser(x, y) ← older(x, y)
- Infer a new proposition:

  wiser(x, y) ← parent(x, y)
- In this case, h is the predicate older(x, y)

Instantiation, and Unification

- During resolution, theorem solvers make inferences based on facts and predicates
- **Definition: Instantiation** is the assignment of variables to values during resolution (replace variables with values)
- **Definition: Unification** is the process that determines the specific instantiations that can be made to variables during a series of simultaneous resolutions

Unification

- A set of terms unify if and only if each term of the set is identical or a step by step application of a sequence of legal substitutions makes them identical. Identical in this sense means
  - they have the same operation name
  - they have the same number of parameters
  - they all have identical terms in corresponding slots
- Loosely, a “term” can be a predicate name, a value, a variable, …

Example

Suppose we have facts

P2(sam, jill)
P2(jack, jill)
P3(suzy, sam)

P3(jack, suzy)

To “resolve” P1(X) :- P2(X, Y), P3(X, Z) for X, we must instantiate X with the *same value (unify it)*, in all three predicates

Instantiation/Unification Example

Given the propositions and predicate below:

speaks(john, french)
speaks(mary, english) and
talkswith(X, Y) ← speaks(X,L), speaks(Y,L), X≠Y

We can infer (using resolution)

talkswith(mary, Y) ← speaks(mary, english), speaks(Y,english), mary≠Y

talkswith(john, Y) ← speaks(john, french), speaks(Y,french), john≠Y

But not

talkswith(mary, john) ← speaks(mary, english), speaks(john,french), mary≠john

Because unification requires all instances of L to be the same.

Prolog Program Elements

- Prolog programs consist of terms (constants, variables, structures)
  - Constant: an <u>atom</u> (horse, dog) or a number or a special character (+, -, =, etc.)
  - Variable: a series of letters, digits, and underscores that begin with an uppercase letter or an underscore.
  - Structure: a predicate with zero or more arguments, written in functional notation (speaks(X,Y))

Relations

- Prolog programs specify *relationships* among objects and properties of objects.
- When we say, "John owns the book", we are declaring the ownership relationship between two objects: John and the book.
- When we ask, "Does John own the book?" we are trying to find out about a relationship.
- Relationships can also rules such as: Two people are sisters **if** they are both female **and** they have the same parents.
- A rule allows us to find out about a relationship even if the relationship isn't explicitly stated as a fact.

A little more on being sisters

- As usual in programming, you need to be a bit careful how you phrase things:
- The following would be better:

A and B are sisters **if**

A and B are both female **and**
they have the same father **and**
they have the same mother **and**
A is not the same as B

Atoms

- Atoms are similar to literals (constants) in a traditional programming language.
- Atoms normally start with lower case letters, variables with upper case. – To override the default naming convention, use single-quotes:
  - ‘Jack’ is an atom, Jack is a variable, jack is an atom by default (no quotes, lower case)
- Numbers in Prolog are integers or reals (although reals aren’t used often in Prolog)
  - Integer format: 1 -375 0
  - Real format (implementation dependent): e.g., -3.3 2.95

Structure

- A structure is a predicate with 0 or more components (arguments) – parent(X, Y).
  - between(X, Y, Z).
  - Date(Day, Month, Year) or Date(1, april, 2023) or Date(Day, july, 2022).
- Structure components can be structures
  - Given Point1(1, 5) and Point2(2, 2) then define Segment(Point1, Point2)
- Generic format: *functor(parameter list)*, where “functor” is similar to a function name in other languages
  - Parameter list: atoms, variables, other structures
  - personRec(name(smith,john), date(28,feb,1963)).
- Functors can be overloaded; the Prolog system uses the arity (number of arguments) of the structure to determine which definition to use.

Facts, Rules, Queries

- Prolog programs are built out of facts and rules; users formulate queries to state the program goals.
- Facts, rules, and queries are composed of the basic language elements (terms) – atoms, variables, and structures.

Programming in Prolog

- declare facts describing explicit relationships between objects and properties objects might have (e.g. Mary likes pizza, grass has_colour green, Fido is_a_dog, Mizuki taught Paul Japanese)
- define rules defining implicit relationships between objects (e.g. the sister rule above) and/or rules defining implicit object properties (e.g. X is a parent if there is a Y such that Y is a child of X).

One then uses the system by:

- asking questions above relationships between objects, and/or about object properties (e.g. does Mary like pizza? is Joe a parent?)

Prolog Facts

- A fact is **a term** followed by **a period**: *speaks (bob, english).* The arguments are atoms.
- Facts are similar to headless Horn clauses. There are built from propositions that are assumed to be true.
- Facts can't include **variables** [it specify Properties of objects, *or* relationships between objects];
- "Dr Turing lectures in course 9020", is written in Prolog as:
  lectures(turing, 9020).

 Notice that:

  - names of properties/relationships begin with lower case letters.
  - the relationship name appears as the first term
  - objects appear as comma-separated arguments within parentheses.
  - A period "." must end a fact.
  - objects also begin with lower case letters. They also can begin with digits (like 9020), and can be strings of characters enclosed in quotes (as in reads(fred, "War and Peace")).

- lectures(turing, 9020). is also called a *predicate*

Rules in Prolog

- a rule consists of three parts: **Head, Neck (if condition) , Body** and **a period.**
- A rule is a term followed by: - and one or more terms, ending in a period:
  term :- $term_1$, $term_2$, …$term_n$.

grandparent (X,Z) :- parent(X, Y), parent(Y, Z).

- Rules are headed Horn clauses.
- The operator (neck) " *:-* " in a rule means *"if"*

Fact/Rule Semantics parent(X, Y). mean "X is the parent of Y", or "Y is the parent of X"?

- The semantics of Prolog rules and facts are supplied by the programmer.
- The Prolog system assumes that the left hand side (LHS) of a rule is true only if all the terms on the RHS are true.
- Use of variables makes rules general (provides a kind of universal quantification.)
  female(X) :- mother(X).
  is equivalent to
  ∀x(mother(x)) ⊃ female(x)
- The commas **(,)**that separate terms on the RHS represent the 'and' operator, so all must be true; in other words, the terms represent a conjunction of conditions.
- To express disjunction (or), use additional rules:
  parent(X,Y) :- mother(X,Y).
  parent(X,Y) :- father(X, Y).
- **Or** … use a **semicolon (;)** to indicate disjunction:
  parent(X,Y) :- mother(X,Y); father(X, Y).

Facts, Rules, & Goal Statements

- Goals (queries) are propositions to be proven true or false, based on facts that are given as part of the program
- Syntax for a goal statement:
  - headless Horn clause: dog(spot).
  - a conjunction of clauses: dog(spot), owner(Who, spot).
  - clauses with several parameters: father(X, Y).

Success and Failure

- A rule *succeeds* when there are instantiations (temporary assignments to variables) for which all right-hand terms are true.
- A rule *fails* if it doesn't succeed.
- Facts always succeed.

```
speaks(allen, russian). speaks(bob,
english). speaks(mary, russian).
speaks(mary, english).
talkswith(X, Y):- speaks(X, L),
speaks(Y, L), X \= Y.
```

- This program has four facts and one rule.

- The rule *succeeds* for any instantiation of its variables in which all the terms on the right of ":-"are simultaneously true; e.g., for the instantiation X=allen, Y=mary, and L=russian.
- For other instantiations, like X=allen and Y=bob, the rule *fails*.

Facts about a hypothetical computer science department:

```
% lectures(X, Y): person X lectures in course Y
lectures(turing, 9020).   lectures(codd, 9311).   lectures(backus, 9021).
lectures(ritchie, 9201).  lectures(minsky, 9414).
lectures(codd, 9314).

% studies(X, Y): person X studies in course Y
studies(fred, 9020).     studies(jack, 9311).
studies(jill, 9314).     studies(jill, 9414).
studies(henry, 9414).    studies(henry, 9314).

%year(X, Y): person X is in year Y
year(fred, 1).   year(jack, 2).
year(jill, 2).   year(henry, 4).
```

Together, these facts form Prolog's *database*.

Queries

- Once we have a database of facts (and, soon, rules) we can ask questions about the stored information.
- A *query* is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interest; e.g., ?- speaks(Who, russian).
- asks for an instantiation of the variable Who for which the query succeeds.
- Other queries:

```
?- talkswith(Who, allen).
?- speaks(allen, L).
?- talkswith(allen, mary).
```

- Suppose we also want to know if Turing lectures in course 9020. We can ask:

% **prolog -s facts03**	facts03 loaded into Prolog
(multi-line welcome message)	"?-" is Prolog's prompt
?- *lectures(turing, 9020).* true.	output from Prolog hold
?- *<control-D>*	down control & press D to
%	leave Prolog

- *Notice that:*

- In SWI Prolog, queries are terminated by a full stop. ○ To answer this query, Prolog consults its database to see if this is a known fact.
- In example dialogues with Prolog, the text in *green italics* is what the user types.

?- *lectures(codd, 9020).*
false.

- if answer is true., the query *succeeded*
- if answer is false., the query *failed*. Note: many early versions of Prolog, including early versions of SWI-Prolog, say No instead of false. See the article on negation in the Prolog dictionary to find out why No. is a more accurate description of what is happening.
- In the latest version of SWI Prolog, it no longer says "No." but says "false." instead.
- The use of lower case for codd is critical.
- Prolog is not being intelligent about this - it would not see a difference between this query and lectures(fred, 9020). or lectures(xyzzy, 9020). though a person inspecting the database can see that fred is a student, not a lecturer, and that xyzzy is neither student nor lecturer.

Variables

- Suppose we want to ask, "What course does Turing teach"?
- This could be written as:
- Is there a course, X, that Turing teaches?
- The variable X stands for an object that the questioner does not know about yet.
- To answer the question, Prolog has to find out the value of X, if it exists.
- As long as we do not know the value of a variable it is said to be *unbound*.
- When a value is found, the variable is said to *bound* to that value.
- The name of a variable must begin with a capital letter or an underscore character, "_".

Variables 2

- To ask Prolog to find the course that Turing teaches, enter this: ?- *lectures(turing, Course).*

Course = 9020 ← output from Prolog

- To ask which course(s) Prof. Codd teaches, we may ask, ?- *lectures(codd , Course).*

Course = 9311 *;* ← type ";" to get next solution

Course = 9314

?-

If Prolog can tell that there are no more solutions, it just gives you the ?- prompt for a new query, as here. If Prolog can't tell, it will let you type ; again, and then if there is no further solution, report false.

- Prolog can find all possible ways to answer a query, unless you explicitly tell it not to (see *cut*, later).

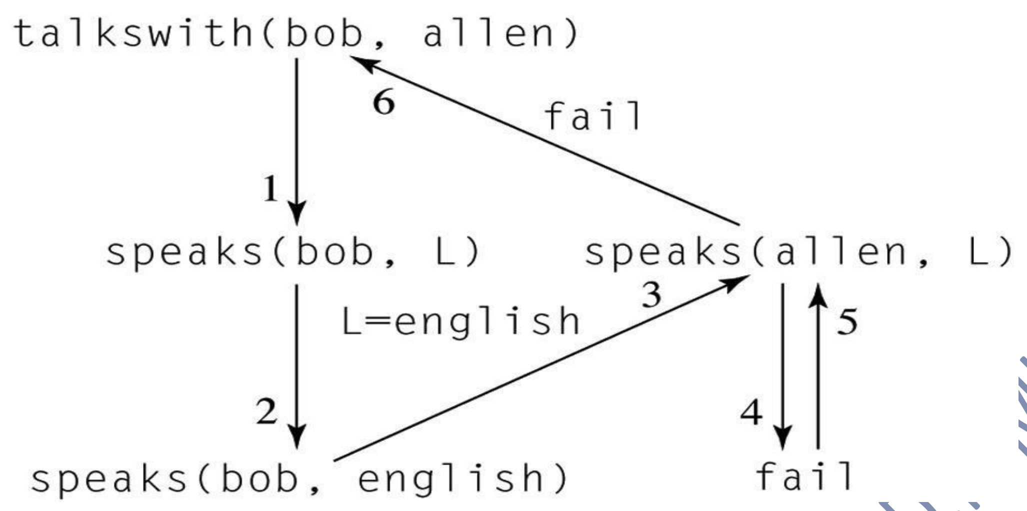
Unification, Evaluation Order, and Backtracking

- To answer a query, examine the rules and facts whose head matches the function in the query.
- e.g.,: to solve speaks(Who,russian). look at – speaks(allen, russian).
  - speaks(bob, english).
  - speaks(mary, russian).
  - speaks(mary, english).
- Possible solutions are considered in order.
- Instantiating Who with allen generates the first match (since the second argument matches the query).

?-talkswith(bob, allen)

speaks(allen,russian). speaks(bob, english). speaks(mary, russian). speaks(mary, english). talkswith(P1,P2) :- speaks(P1,L), speaks(P2,L), P1\=P2.	• *Instantiate and unify* variables in the rule with facts; *i.e.*, find values for P1, P2, and L that satisfy all 3 subgoals. P1 = bob, P2 = allen • Can we find a value of L? • Solve subgoals left to right in rule, work top to bottom in facts • In case of failure, back-track to nearest subgoal

Attempting to Satisfy the Query talkswith (bob, allen)



Conjunctions of Goals in Queries

- How do we ask, "Does Turing teach Fred"?
- This means finding out if Turing lectures in a course that Fred studies.

  ?- *lectures(turing, Course), studies(fred, Course).*
- i.e. "Turing lectures in course, Course **and** Fred studies (the same) Course".
- The question consists of two *goals*.
- To answer this question, Prolog must find a single value for Course that satisfies both goals.
- Read the comma, ",", as **and**.
- However, note that Prolog will evaluate the two goals left-to-right. In pure logic, P1 ∧ P2 is the same as P2 ∧ P1. In Prolog, there is the practical consideration of which goal should be evaluated first - the code might be more efficient one way or the other. In particular, in "P1, P2", if P1 fails, then P2 does not need to be evaluated at all. This is sometimes referred to as a "conditional-and".

Disjunctions of Goals in Queries

- What about **or** (i.e. disjunction)? It turns out explicit **or**s aren't needed much in Prolog

There is a way to write **or**: (";")

- The reason **or**s aren't needed much is that head :- body1. head :- body2. has the same effect as head :- body1 ; body2.
- Avoid using ; if you can, at least until you have learned how to manage without it. While some uses of ; are harmless, others can make your code hard to follow.
- To reinforce this message, you will be **penalised** if you use the **or** operator ; in the first

(Prolog) assignment in COMP9414. This prohibition means you can't use the … -> … ; … construct either, in the first assignment. The … -> … ; … construct is not taught in COMP9414, but in the past, some people have found out about it.

Backtracking in Prolog

- Who does Codd teach?
- ?- *lectures(codd, Course), studies(Student, Course).*

Course = 9311

Student = jack *;*

Course = 9314

Student = jill *;*

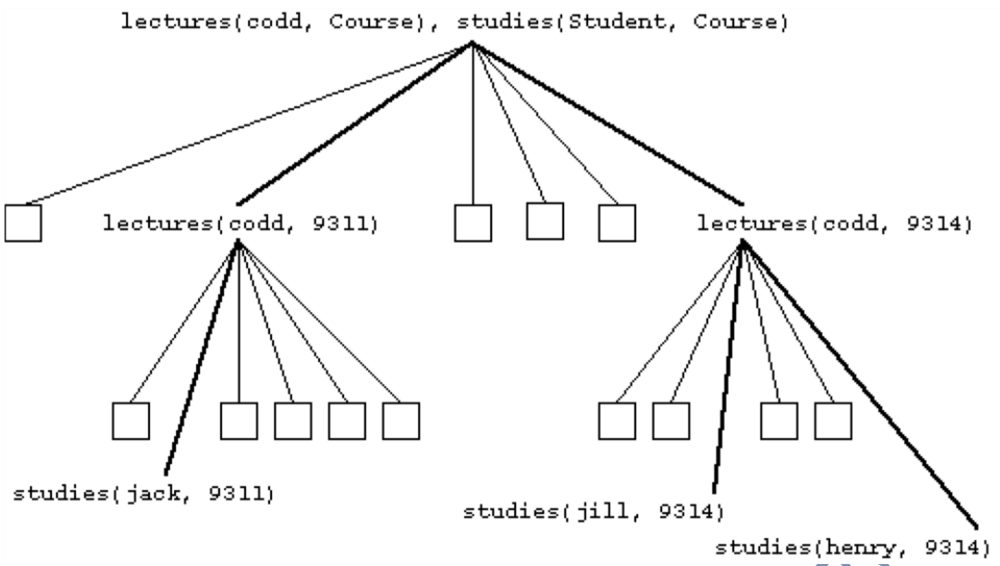
Course = 9314

Student = henry *;*

- Prolog solves this problem by proceeding left to right and then *backtracking*. When given the initial query, Prolog starts by trying to solve

  lectures(codd, Course)
- There are six lectures clauses, but only two have codd as their first argument.
- Prolog uses the first clause that refers to codd: lectures(codd, 9311).
- With Course = 9311, it tries to satisfy the next goal, studies(Student, 9311).
- It finds the fact studies(jack, 9311). and hence the first solution: (Course = 9311, Student = jack)
- After the first solution is found, Prolog retraces its steps up the tree and looks for alternative solutions.
- First it looks for other students studying 9311 (but finds none).
- Then it ○ backs up
  - rebinds Course to 9314,
  - goes down the lectures(codd, 9314) branch
  - tries studies(Student, 9314), ○ finds the other two solutions: (Course = 9314, Student = jill) and (Course = 9314, Student = henry).

To picture what happens when Prolog tries to find a solution and backtracks, we draw a "proof tree":



Clause Syntax

- ":-" means "if" or "is implied by". Also called the *neck* symbol.
- The left hand side of the neck is called the *head*.
- The right hand side of the neck is called the *body*.
- The comma, ",", separating the goals, stands for *and*.
- Another rule, using the *predefined predicate* ">".

```
    more_advanced(S1, S2) :-
    year(S1, Year1),
    year(S2, Year2),        Year1
> Year2.
```

Tracing Execution

```
    more_advanced(S1, S2) :-
    year(S1, Year1),
    year(S2, Year2),
    Year1 > Year2.
```

?- *trace*. true.	

[trace] ?- *more_advanced(henry, fred).*	bind S1 to henry, S2 to fred
Call: more_advanced(henry, fred) ? *	test 1st goal in body of rule
Call: year(henry, _L205) ?	succeeds, binds Year1 to 4
Exit: year(henry, 4) ?	test 2nd goal in body of rule
Call: year(fred, _L206) ?	succeeds, binds Year2 to 1
Exit: year(fred, 1) ?	test 3rd goal: Year1 > Year2
^ Call: 4>1 ?	succeeds succeeds
^ Exit: 4>1 ?	
Exit: more_advanced(henry, fred) ?	
true.	
[debug] ?- *notrace.*	

* The ? is a prompt. Press the return key at end of each line of tracing. Prolog will echo the <return> as creep, and then print the next line of tracing. The "creep"s have been removed in the table above, to reduce clutter.

true., false., or true

- Sometimes, Prolog says true instead of true. (i.e. no full-stop after true).
- Prolog does this when it believes it may be able to prove that the query is true in more than one way (and there are no variables in the query, that it can report bindings for).
- Example: suppose we have the following facts and rule: bad_dog(fido). bad_dog(Dog) :- bites(Dog, Person),

```
  is_person(Person),
is_dog(Dog). bites(fido,
postman).
is_person(postman).
is_dog(fido).
```

There are two ways to prove bad_dog(fido): (a) it's there as a fact; and (b) it can be proven using the bad_dog rule:

?- *bad_dog(fido).*

true *;* true.

The missing full-stop prompts us to type ; if we want to check for another proof. The true. that follows means that a second proof *was* found. Alternatively, we can just press the "return" key if we are not interested in whether there is another proof.

Library Database Example

- A database of books in a library contains facts of the form : book(CatalogNo, Title, author(Family, Given)).
- libmember(MemberNo, name(Family, Given), Address).
- loan(CatalogNo, MemberNo, BorrowDate, DueDate).
- A member of the library may borrow a book.
- A "loan" records:

    - the catalogue number of the book
    - the number of the member
    - the date on which the book was borrowed
    - the due date
- Dates are stored as structures:
  date(Year, Month, Day)
- e.g. date(2022, 6, 16) represents 16 June 2022.
- which books has a member borrowed?

```
 borrowed(MemFamily, Title, CatalogNo) :-    libmember(MemberNo,
name(MemFamily, _), _),
   loan(CatalogNo, MemberNo, _, _),
book(CatalogNo, Title, _).
```

- The underscore or "don't care" variables (_) are used because for the purpose of this query we don't care about the values in some parts of these structures.

Comparing Two Terms

- we would like to know which books are overdue; how do we compare dates?

```
%later(Date1, Date2) if Date1 is after Date2: later(date(Y, M, Day1), date(Y, M, Day2)) :-
    Day1 > Day2.
later(date(Y, Month1, _), date(Y, Month2, _)) :-          Month1 > Month2.
later(date(Year1, _, _), date(Year2, _, _)) :-       Year1 > Year2.
```

- This rule has three clauses: in any given case, only one clause is appropriate. They are tried in the given order.
- This is how disjunction (**or**) is often achieved in Prolog. In effect, we are saying that the first date is later than the second date if Day1 > Day2 and the Y and M are the same, **or** if the Y is the same and Month1 > Month2, **or** if Year1 > Year2.
- *Footnote*: if the year and month are the same, then the heads of all three rules match, and so, while the first rule is the appropriate one, all three will be tried in the course of backtracking. However, the condition "Month1 > Month2" in the second rule means that it will fail in this case, and similarly for the third rule.
- In the code for later, again we are using the comparison operator ">"

```
% Facts
book(101, 'Introduction to Prolog', author('Smith', 'John')).
book(102, 'Logic Programming Basics', author('Jones', 'Alice')).
book(103, 'Artificial Intelligence', author('Brown', 'David')).

libmember(001, name('Johnson', 'Mary'), '123 Main Street').
libmember(002, name('Wilson', 'Bob'), '456 Oak Avenue').
libmember(003, name('Taylor', 'Emily'), '789 Elm Lane').

loan(101, 001, date(2022, 6, 1), date(2022, 7, 1)).
loan(102, 002, date(2022, 5, 1), date(2022, 6, 1)).
loan(103, 003, date(2022, 7, 1), date(2022, 8, 1)).

% Rules
borrowed(MemFamily, Title, CatalogNo) :-
    libmember(MemberNo, name(MemFamily, _), _),
    loan(CatalogNo, MemberNo, _, _),
    book(CatalogNo, Title, _).

% Comparing Two Dates
later(date(Y, Month1, Day1), date(Y, Month2, Day2)) :-
    Day1 > Day2,
    Month1 =:= Month2.
later(date(Year1, Month1, _), date(Year2, Month2, _)) :-
    Year1 > Year2.
later(date(Year, Month, _), date(Year, Month, Day1)) :-
    Day1 > 0.

% Finding overdue books
overdue_books(Title, CatalogNo, DueDate) :-
    loan(CatalogNo, _, _, DueDate),
    get_current_date(CurrentDate),
    later(CurrentDate, DueDate),
    book(CatalogNo, Title, _).

% Example usage
% To find books borrowed by a member:
% ?- borrowed('Johnson', Title, CatalogNo).
% This will find books borrowed by the member with the family name 'Johnson'.

% To find overdue books:
% ?- overdue_books(Title, CatalogNo, DueDate).
% This will find books that are overdue based on the current date.
```

- More complex arithmetic expressions can be arguments of comparison operators - e.g. X + Y >= Z * W * 2.
- The available *numeric* comparison operators are:

Operator	**Meaning**	**Syntax**
>	greater than	Expression1 > Expression2
<	less than	Expression1 < Expression2
>=	greater than or equal to	Expression1 >= Expression2
=<	less than or equal to	Expression1 =< Expression2
=:=	equal to	Expression1 =:= Expression2
=\=	not equal to	Expression1 =\= Expression2

- All these numerical comparison operators evaluate both their arguments. That is, they evaluate Expression1 and Expression2.

Overdue Books

```
% overdue(Today, Title, CatalogNo, MemFamily): %
given the date Today, produces the Title, CatalogNo, %
and MemFamily of all overdue books.

overdue(Today, Title, CatalogNo, MemFamily) :-
loan(CatalogNo, MemberNo, _, DueDate),
  later(Today, DueDate),   book(CatalogNo, Title,
_),   libmember(MemberNo, name(MemFamily, _),
_).
```

Due Date

- Assume the loan period is one month:

```
due_date(date(Y, Month1, D),
date(Y, Month2, D)) :-
        Month1 < 12,
            Month2 is Month1 + 1.
due_date(date(Year1, 12, D),
date(Year2, 1, D)) :-
Year2 is Year1 + 1.
```

The is operator

- The right hand argument of is must be an arithmetic expression that can be evaluated right now (no unbound variables).
- This expression is evaluated and bound to the left hand argument.

- **"is"** is not a C-style assignment statement:
  - X is X + 1 won't work!
  - except via backtracking, variables can only be bound once, using is or any other way
- "=" does not cause evaluation of its arguments:

?- *X = 2, Y = X + 1.*
X = 2
Y = 2+1
	?- *X = 2, Y is X + 1.*
	X = 2
	Y = 3

- Use **is** if and only if you need to evaluate something: X is 1 BAD! - nothing to evaluate X = 1 GOOD!
- To reinforce the point about the meaning of **is**, you will be penalised in the first Prolog assignment if you use it where it is not needed.

Order of goals with is

- Order of goals matters with **is.**

  Variables on the RHS of is *must* be instantiated at the time the is goal is tried by Prolog. This is why the following example fails:

  ?- X is Y + 1, Y = 3.

  ERROR: is/2: Arguments are not sufficiently instantiated

  vs

  ?- Y = 3, X is Y + 1.

  Y = 3,

  X = 4.

is, = and =:=

- You can see the differences between these three Prolog constructs from the following example Prolog queries:

?- X =:= 3+2. ERROR: =:=/2: Arguments are not sufficiently instantiated	X is not currently bound, so can't be evaluated.
?- X = 3+2. X = 3+2.	= doesn't evaluate, so X is bound to 3+2.
?- X is 3+2. X = 5.	is *does* evaluate its right-hand side.
?- 4+1 is 3+2. false.	3+2 is evaluated to 5. 4+1 is not evaluated. So 4+1 is different from 5.
?- 4+1=3+2. false.	Neither side is evaluated by =. The two expressions are different.

?- 4+1 =:= 3+2. true.	Both sides are evaluated by =:=

= is used for matching, so a more appropriate use would be:

?- likes(mary, X) = likes(Y, pizza).

X = pizza,

Y = mary.

Write a prolog program to sum two numbers: Option 1:

```
Clause: sum(X,Y,
SUM):- SUM is X+Y.
Goal: sum(3,5,S).
S=8.
```

Option 2:

```
sum(X,Y):- sum is
X+Y, write(sum).
Goal: sum(3,5,S).
S=8.
```

Option 3:

```
Sum:- readint(X),
readint(Y), sum is
X+Y, write (sum).
Goal: sum(3,5,S).
S=8.
```

Recursive Programs

- Recursion in any language is a function that call itself until a given goal has been succeeded
- In prolog, recursion appear when a predicate contain a goal that refers to itself.
- There are two types:

  Tail Recursion: in which the recursive call is always made Justin the last step before the procedure exits.

  Examples:

  Write a prolog program to find the factorial of 5. i.e. 5!

```
Domain : I=Integer Predicate:
fact(I,I,I).
Clauses:
fact(I,F,F):-!. fact(N,F,R):- F1 is F*N, N1 is N-1,
fact(N1,F1,R).
```

Goal: ? - fact(5,I,F).

Output : F=120.

Non-Trail Recursion : is recursion in which the recursion is not the last step in the procedure call.

```
fact(0,1). fact(1,1).
fact(N,F):- N1 is N-1, fact(N1,F1),  F is N*F1.
```

Goal : ?- fact(5,F).

Output : F=120.

Write a prolog program to print number from 1-20.

```
count(20):-!.
count(X):- write (X), X1 is X+1, count(X1).
Goal: count(1).
Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Write a program to find the power of any number using tail recursion.

```
power(_, 0, P,P):-!. power(X, Y, Z,P):- Z1 is Z*X, Y1 is Y-1, power(X, Y1, Z1,P),!.
Goal : power(5, 2, 1,P) Output: P=25.
```

Using non-trail recursion

```
power(_, 0, 1):-!. power(X, Y,P):- Y1 is Y-1, power(X, Y1,P1), P is P1*X.
Goal : power(5, 2, P) Output: P=25.
```

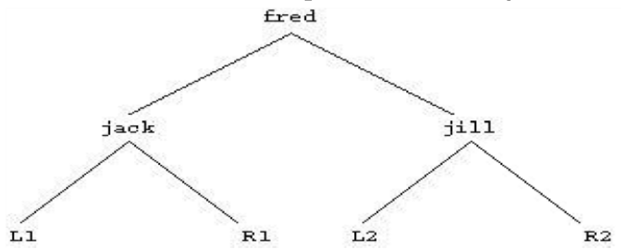
Write a program to read and write a number of characters until the input character is equal to '#'.

```
Clause:
Repeat.
Repeat: repeat.
Typewriter :- repeat, readchar(C), write(C),nl, C is '#',!.
```

Binary Trees

- In the library database example, some complex terms contained other terms, for example, book contained name.
- The following term also contains another term, this time one similar to itself:
- tree(tree(L1, jack, R1), fred, tree(L2, jill, R2))

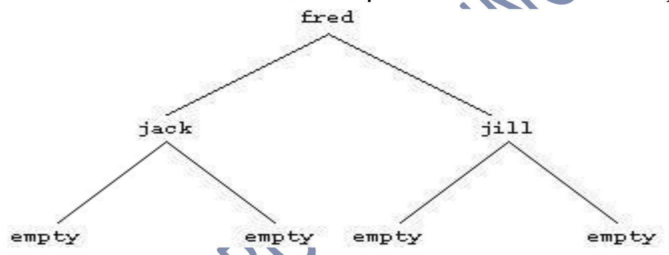
- The variables L1, L2, R1, and R2 should be bound to sub-trees (this will be clarified shortly).
- A structure like this could be used to represent a "binary tree" that looks like:



- Binary because each "node" has two branches (our backtrack tree before had many branches at some nodes)

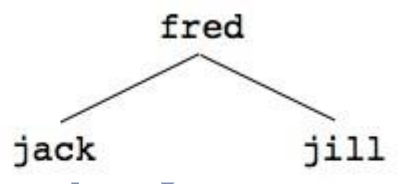
Recursive Structures

- A term that contains another term that has the same principal functor (in this case tree) is said to be recursive.
- Biological trees have leaves. For us, a *leaf* is a node with two empty branches:



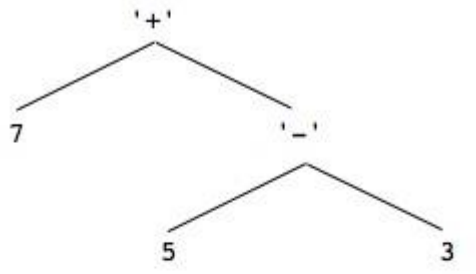
- empty is an arbitrary symbol to represent the empty tree. In full, the tree above would be:
  tree(tree(empty, jack, empty), fred, tree(empty, jill, empty))

 Usually, we wouldn't bother to draw the empty nodes:



Another Tree Example

```
tree(tree(empty, 7, empty),
    '+',
    tree(tree(empty, 5, empty),
        '-',
        tree(empty, 3, empty)))
```



- A binary tree is either empty or contains some data and a left and right subtree that are also binary trees.
- In Prolog we express this as:

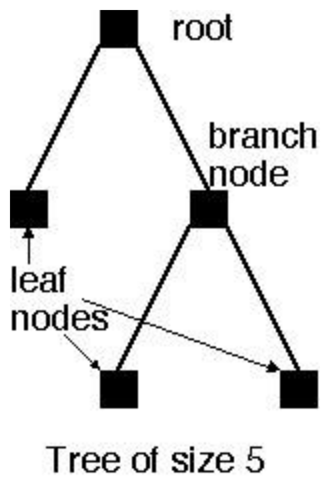
  is_tree(empty). trivial branch

  is_tree(tree(Left, Data, Right)) :- recursive branch
  is_tree(Left), some_data(Data),
  is_tree(Right).
- A non-empty tree is represented by a 3-arity term.
- Any recursive predicate must have:
  - (at least) one **recursive branch/rule** (or it isn't recursive :-)) and
  - (at least) one non-recursive or **trivial branch** (to stop the recursion going on for ever).

  The example at the heading "An Application of Lists", below, will show how the recursive branch and the trivial branch work together. However, you probably shouldn't try to look at it until we have studied lists.
- Let us define (or measure) the size of tree (i.e. number of nodes): tree_size(empty, 0). tree_size(tree(L, _, R), Total_Size) :- tree_size(L, Left_Size), tree_size(R, Right_Size),

  Total_Size is

  Left_Size + Right_Size + 1.



- The size of an empty tree is zero.
- The size of a non-empty tree is the size of the left sub-tree plus the size of the right sub-tree plus one for the current tree node. ▯ The data does not contribute to the total size of the tree.
- Recursive data structures need recursive programs. A recursive program is one that refers to itself, thus, tree_size contains goals that call for the tree_size of smaller trees **Lists**
- A list may be nil (i.e. empty) or it may be a term that has a head and a tail ▯ The head may be any term or atom.
- The tail is another list.
- We could define lists as follows: is_list(nil).

is_list(list(Head, Tail)) :-
is_list(Tail).

- A list of numbers [1, 2, 3] would look like: list(1, list(2, list(3, nil)))
- This notation is understandable but clumsy. Prolog doesn't actually recognise it, and in fact uses . instead of list and [] instead of nil. So Prolog would recognise .(1, .(2, .(3, []))) as a list of three numbers. This is briefer but still looks strange, and is hard to work with.
- Since lists are used so often, Prolog in fact has a special notation that encloses the list members in square brackets:

[1, 2, 3] = .(1, .(2, .(3, [])))

?- *X = .(1, .(2, .(3, [])))*.
X = [1, 2, 3]

List Constructor |

- Within the square brackets [ ], the symbol | acts as an operator to construct a list from an item and another list.

?- *X = [1 | [2, 3]]*.
X = [1, 2, 3].
?- *Head = 1 , Tail = [2, 3], List = [Head | Tail]*.
List = [1, 2, 3].

Examples of Lists and Pattern Matching

?- *[X, Y, Z] = [1, 2, 3]*. Match the terms on either side of =
X = 1
Y = 2
Z = 3

| separates head from tail of list. ?-
[X | Y] = [1, 2, 3].
X = 1 So [First | Rest] is the usual way
Y = [2, 3] of writing .(First, Rest) in Prolog

?- *[X | Y] = [1]*. The empty list is written as [] Lists
X = 1 "end" in an empty list!
Y = [] Note that [1] is a list with one element.

The first several elements of the list can be selected before matching the tail:
?- *[X, Y | Z] = [fred, jim, jill, mary]*.
X = fred Must be at least two elements Y = jim in the list
on the right.

Z = [jill, mary]

Complex List Matching

?- *[X | Y] = [[a, f(e)], [n, m, [2]]].*

X = [a, f(e)]
Y = [[n, m, [2]]]
Notice that Y is shown with an extra pair of brackets: Y is the tail of the entire list: [n, m, [2]] is the sole element of Y.

List Membership

- A term is a member of a list if ○ the term is the same as the head of the list, or ○ the term is a member of the tail of the list.
- In Prolog:

```
                    trivial branch:
member(X, [X | _]).
                         a rule with a head but no body
member(X, [_ | Y]) :-   recursive branch
    member(X, Y).
```

- The first rule has the same effect as: member(X, [Y|_]):- X = Y.
  The form member(X, [X|_]). is preferred, as it avoids the extra calculation.
- Member is actually predefined in Prolog. It is a built-in predicate. There are quite a few built-in predicates in Prolog

```
% length(List, LengthOfList)
% binds LengthOfList to the number of elements in List.
length([OnlyMember], Length) :-
      Length = 1.
length([First | Rest], Length) :-
 length(Rest, LengthOfRest),  Length is
LengthOfRest + 1.
```

This works, but involves an unnecessary unification. It is better for the base case to be length([OnlyMember], 1).
In effect, we take the original version of the base case, and replace Length, in the head of the rule, with the thing that Length is = to. Programmers who fail to do this are usually still thinking procedurally.

Programming Principles for Recursive Structures

- Only deal with one element at a time.

- Believe that the recursive program you are writing has already been written. In the definition of member, we are already assuming that we know how to find a member in the tail.
- Write definitions, not programs!
    - If you are used to writing programs for conventional languages, then you are used to giving instructions on how to perform certain operations.
    - In Prolog, you define relationships between objects and let the system do its best to construct objects that satisfy the given relationship.

Concatenating Two Lists

- Suppose we want to take two lists, like [1, 3] and [5, 2] and concatenate them to make [1, 3, 5, 2]
- The header comment is:

% concat(List1, List2, Concat_List1_List2)

% Concat_List1_List2 is the concatenation of List1 & List2 There are two rules:

- First, the trivial branch: concat([], List2, List2).
- Next, the recursive branch:

  concat([Item | Tail1], List2, [Item | Concat_Tail1_List2]) :- concat(Tail1, List2, Concat_Tail1_List2).

- For example, consider

  ?- concat([1], [2], [1, 2]).
  By the recursive branch:

- concat([1 | []], [2], [1 | [2]]) :-concat([], [2], [2]). and concat([], [2], [2]) holds because of the trivial branch.
- The entire program is:

```
% concat(List1, List2, Concat_List1_List2):
%   Concat_List1_List2 is the concatenation of List1 & List2
concat([], List2, List2). concat([Item | Tail1], List2, [Item |
Concat_Tail1_List2]) :-       concat(Tail1, List2,
Concat_Tail1_List2).
```

An Application of Lists

- Find the total cost of a list of items:

  Cost data:

```
cost(cornflakes, 230).
cost(cocacola, 210).
cost(chocolate, 250).
cost(crisps, 190).
```

- Rules:

```
total_cost([], 0).              % trivial branch
total_cost([Item|Rest], Cost) :-  % recursive branch
cost(Item, ItemCost),     total_cost(Rest,
CostOfRest),     Cost is ItemCost + CostOfRest.
```

Sample query:

?- *total_cost([cornflakes, crisps], X).*

X = 420

Tracing total_cost ?-
trace.

```
true.
[trace]  ?- total_cost([cornflakes, crisps], X).
   Call: (7) total_cost([cornflakes, crisps], _G290) ? creep
   Call: (8) cost(cornflakes, _L207) ? creep
   Exit: (8) cost(cornflakes, 230) ? creep
   Call: (8) total_cost([crisps], _L208) ? creep
   Call: (9) cost(crisps, _L228) ? creep
   Exit: (9) cost(crisps, 190) ? creep
   Call: (9) total_cost([], _L229) ? creep
   Exit: (9) total_cost([], 0) ? creep
^  Call: (9) _L208 is 190+0 ? creep
^  Exit: (9) 190 is 190+0 ? creep
   Exit: (8) total_cost([crisps], 190) ? creep
^  Call: (8) _G290 is 230+190 ? creep
^  Exit: (8) 420 is 230+190 ? creep
   Exit: (7) total_cost([cornflakes, crisps], 420) ? creep

X = 420

[debug]  ?- notrace.
```

Modifying total_cost

This is an *optional* homework exercise.

What happens if we change the recursive branch rule for total_cost as shown below?

```
total_cost([Item|Rest], Cost) :-     total_cost(Rest, CostOfRest),     cost(Item,
ItemCost),
    Cost is ItemCost + CostOfRest.
```

The second and third lines have been swapped around.

You'll find that the rule still works. Try tracing the new version of this rule, work out what happens differently.

Which version do you find easier to understand? Why do think this is the case?

Another list-processing procedure

- The next procedure removes duplicates from a list.
- It has *three rules*. This is an example of a common list-processing *template*.
- Algorithm: ○ If the list is empty, there's nothing to do.
  - ○ If the first item of the list is a member of the rest of the list, then discard it, and remove duplicates from the rest of the list. ○ Otherwise, keep the first item, and again, remove any duplicates from the rest of the list.

```
% remove_dups(+List, -NewList):
% New List isbound to List, but with duplicate items removed. remove_dups([],
[]).
remove_dups([First | Rest], NewRest) :-
member(First, Rest), remove_dups(Rest,
NewRest). remove_dups([First | Rest], [First |
NewRest]) :- not(member(First, Rest)),
remove_dups(Rest, NewRest).
```

```
?- remove_dups([1,2,3,1,3,4], X).

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

false.
```

- Note the use of not to negate a condition. An alternative to not is \+.

Singleton Variables

- If Prolog finds a variable name that you only use once in a rule, it assumes that it may be a spelling mistake, and issues a **Warning** about a "singleton variable" when you load the code:

```
% prolog -q -s mycode.pl
Warning: .../mycode.pl:4:
                Singleton variables: [Item]
```

The -q means "quiet" - i.e. don't print the SWI Prolog welcome message. This way, any warnings are easier to notice.

- Here is the code that produced this (with line numbers added): 1 % count(Item, List, Count) counts the number of times the 2 % Item occurs in the List, and binds Count to that number.

3

4 count(Item, [], 0).

5 count(Item, [Item | Rest], Count) :- 6
count(Item, Rest, RestCount), 7 Count is
RestCount + 1.

8 count(Item, [Other | Rest],
Count) :-

9 not(Item = Other), 10
count(Item, Rest, Count).

To suppress the warning, put an _ in front of the word Item on line 4 (only). This makes it "don't care" variable. Check for the possible spelling error, first!

4 count(_Item, [], 0).

.

Controlling Execution

The Cut Operator/function (!)

- Sometimes we need a way to prevent Prolog finding all solutions, i.e. a way to stop backtracking.
- The cut operator, written !, is a built-in goal that prevents backtracking.

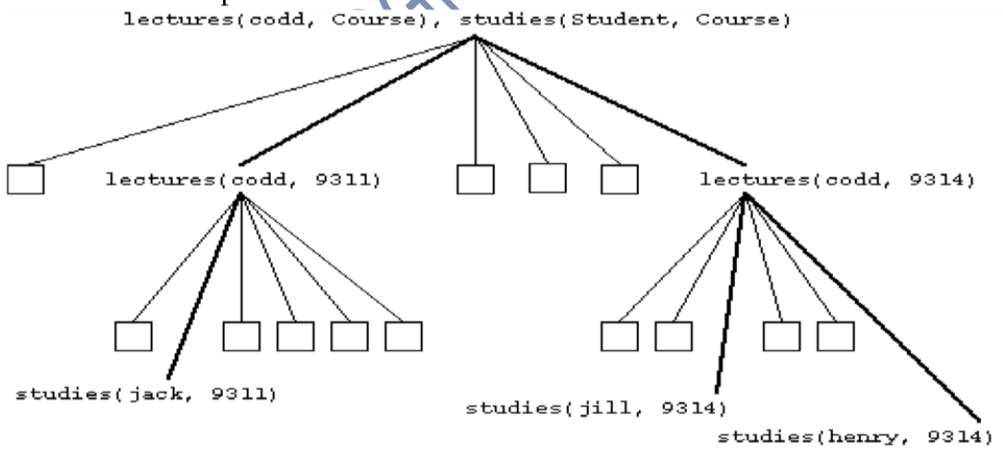
- It turns Prolog from a nice declarative language into a hybrid monster.
- Use cuts sparingly and with a sense of having sinned.

```
% Using cut
no(5):-!.
no(I).
no(10).

| ?- no(X).

X = 5

yes
```

Using cut in the end of the rule.

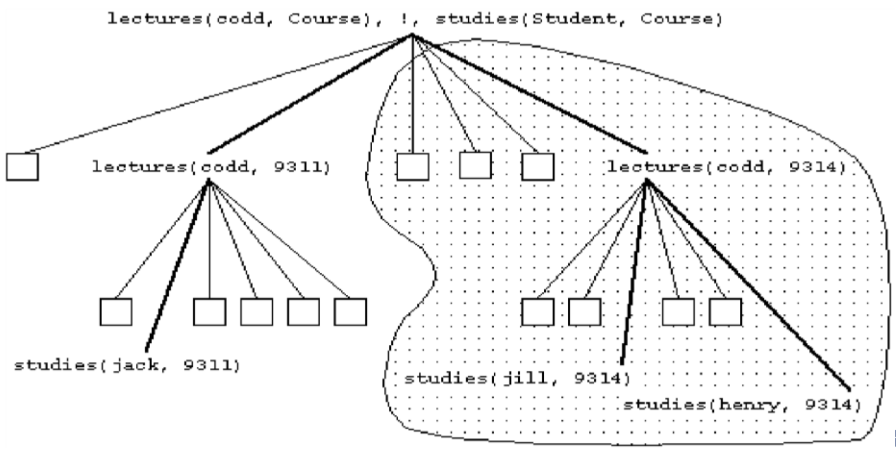
```
a(10).
a(20).
b(a).
b(c).
c(X,Y):- a(X), b(Y), !.

| ?- c(X,Y).
X = 10
Y = a

yes
```

Recall this example:



Cut Prunes the Search Tree

- If the goal(s) to the right of the cut fail then the entire clause fails and the the goal that caused this clause to be invoked fails.



- In particular, alternatives for Course are not explored.

Cut Prunes the Search Tree 2

- Another example: using the facts03 database, try ?- *lectures(codd, X).* X = 9311 *;* X = 9314.

?- *lectures(codd, X), !* .

X = 9311.

- The cut in the second version of the query prevents Prolog from backtracking to find the second solution.

Using cuts in later to improve efficiency

Recall the code for later:

```
later(date(Y, M, D1), date(Y, M, D2)) :- D1 > D2.
later(date(Y, M1, _), date(Y, M2, _)) :- M1 > M2.
later(date(Y1, _, _), date(Y2, _, _)) :- Y1 > Y2.
```

We note that if year and month are the same, all three rules are tried while backtracking. This could be prevented by adding cuts:

```
later(date(Y, M, D1), date(Y, M, D2)) :- D1 > D2, !.
later(date(Y, M1, _), date(Y, M2, _)) :- M1 > M2, !.
later(date(Y1, _, _), date(Y2, _, _)) :- Y1 > Y2.
```

This would increase efficiency by eliminating unnecessary backtracking, though it is doubtful if it would be worth bothering about, unless you actually have code that is running too slowly. In that case you should first do an analysis of where the time is being spent, before putting in cuts everywhere!

In other cases, adding cuts of this sort to multi-rule procedures might be a useful (if lazy) way of ensuring that only one rule is used in a particular case. Unless it makes the code very clumsy, it is better to use and rely on "condition" goals in each rule (like M1 > M2 in the second rule for later) to specify the case in which it is appropriate. More examples of this are below.

Another cut example

- max, without cut:

```
% max(A, B, C) binds C to the larger of A and B.
max(A, B, A) :-    A > B.
max(A, B, B) :- A =< B.
```

- max, with cut:

```
max(A, B, A) :-
A > B,
  !. max(A, B,
B).
```

- The first version has a negated test in the second rule (=< vs >). The second version substitutes a cut in the first rule for the negated test.
- Remember, no cuts in the first assignment unless they are essential! Hint: the first assignment can be done without cuts.

PROGRAMS

TOPIC 6: KNOWLEDGE REPRESENTATION AND REASONING

Introduction

- KR and Reasoning is the field of AI that focuses on designing computer representations that capture information about the world that can be used for reasoning, inference, decision-making, and problem-solving.
- It is dedicated to representing information about the world in a form that a computer system can utilize to solve complex tasks such as diagnosing a medical condition or having a dialog in a natural language.
- The justification for KR is that conventional procedural code is not the best formalism to use to solve complex problems.
- KR makes complex software easier to define and maintain than procedural code and can be used in expert systems
- incorporates findings from psychology about how humans solve problems and represent knowledge in order to design formalisms that will make complex systems easier to design and build
- KR goes hand in hand with automated reasoning because one of the main purposes of explicitly representing knowledge is to be able to reason about that knowledge, to make inferences, assert new knowledge, etc.
- Virtually all knowledge representation languages have a reasoning or inference engine as part of the system
- KR and reasoning also incorporates findings from logic to automate various kinds of reasoning, such as the application of rules or the relations of sets and subsets
- Examples of knowledge representation formalisms include: semantic nets, systems architecture, frames, rules, and ontologies.
- Examples of automated reasoning engines include: inference engines, theorem provers, and classifiers

Types of Knowledge:

- Declarative Knowledge: Describes facts about the world (e.g., "The sky is blue").
- Procedural Knowledge: Describes how to perform specific tasks or actions (e.g., "How to ride a bike").
- Meta Knowledge: Describes knowledge about knowledge, including uncertainty, time, and context.

Expressivity and Practicality in KR

- A key trade-off in the design of a knowledge representation formalism is that between expressivity and practicality
- The ultimate knowledge representation formalism in terms of expressive power and compactness is First Order Logic (FOL).

- FOL drawbacks as a knowledge representation formalism - ease of use and practicality of implementation – FOL can be intimidating even for many software developers
- The issue of practicality of implementation is that FOL in some ways is too expressive.
- With FOL it is possible to create statements (e.g. quantification over infinite sets) that would cause a system to never terminate if it attempted to verify them.
- IF-THEN rules provide a subset of FOL but a very useful one that is also very intuitive.
- a subset of FOL can be both easier to use and more practical to implement - a driving motivation behind rule-based expert systems.
- A KR is most fundamentally a surrogate, a substitute for the thing itself, used to enable an entity to determine consequences by thinking rather than acting, i.e., by reasoning about the world rather than taking action in it.
- It is a set of ontological commitments, i.e., an answer to the question: In what terms should I think about the world?
- It is a fragmentary theory of intelligent reasoning, expressed in terms of three components:
- the representation's fundamental conception of intelligent reasoning; • the set of inferences the representation sanctions; and
- the set of inferences it recommends.
- It is a medium for pragmatically efficient computation, i.e., the computational environment in which thinking is accomplished
- It is a medium of human expression, i.e., a language in which we say things about the world.

KR and Semantic Web

- Knowledge representation and reasoning are a key enabling technology for the Semantic web
- Languages based on the Frame model with automatic classification provide a layer of semantics on top of the existing Internet.
- Rather than searching via text strings as is typical today it will be possible to define logical queries and find pages that map to those queries
- The Semantic web integrates concepts from knowledge representation and reasoning with markup languages based on XML.
- The Resource Description Framework (RDF) provides the basic capabilities to define knowledge-based objects on the Internet with basic features such as Is-A relations and object properties.
- The Web Ontology Language (OWL) adds additional semantics and integrates with automatic classification reasoners.

Ontology engineering and Ontology language

- In the early years of knowledge-based systems the knowledge-bases were fairly small.

- The knowledge-bases that were meant to actually solve real problems rather than do proof of concept demonstrations needed to focus on well defined problems
- As knowledge-based technology scaled up the need for larger knowledge bases and for modular knowledge bases that could communicate and integrate with each other became apparent
- hence the rise to the discipline of ontology engineering, designing and building large knowledge bases that could be used by multiple projects
- Ontologies provide a formal and explicit specification of a shared conceptualization within a specific domain. They define classes, relationships, and properties to represent the structure of knowledge in a domain.
- A number of ontology languages have been developed. Most are declarative languages, and are either frame languages, or are based on first-order logic(e.g., logic, LISP, etc.)

Reasoning under uncertainty

- Many reasoning systems provide capabilities for reasoning under uncertainty - is important when building situated reasoning agents which must deal with uncertain representations of the world.
- Common approaches to handling uncertainty include the use of:
- certainty factors
- probabilistic methods such as Bayesian inference or Dempster–Shafer theory
- multi-valued (‘fuzzy’) logic and
- various connectionist approaches

Types of reasoning system

- **Constraint solvers**
- they solve constraint satisfaction problems (CSPs).
- They support constraint programming.
- A constraint is a condition which must be met by any valid solution to a problem
- Constraints are defined declaratively and applied to variables within given domains

- **Theorem provers**
- they use automated reasoning techniques to determine proofs of mathematical theorems.
- They may also be used to verify existing proofs
- In addition to academic use, typical applications of theorem provers include verification of the correctness of integrated circuits, software programs, engineering designs,

- **Logic programs** (LPs)
- are software programs written using programming languages whose primitives and expressions provide direct representations of constructs drawn from mathematical logic.

- An example of a general-purpose logic programming language is Prolog.
- LPs represent the direct application of logic programming to solve problems.
- Logic programming is characterized by highly declarative approaches based on formal logic, and has wide application across many disciplines.

- **Rule engines**
- they represent conditional logic as discrete rules. Rule sets can be managed and applied separately to other functionality.
- They have wide applicability across many domains.
- Many rule engines implement reasoning capabilities.
- A common approach is to implement production systems to support forward or backward chaining.

- **Deductive classifier**
- they arose slightly later than rule-based systems and were a component of a new type of artificial intelligence knowledge representation tool known as frame languages.
- A frame language describes the problem domain as a set of classes, subclasses, and relations among the classes.
- similar to the object-oriented model - Unlike object-oriented models however, frame languages have a formal semantics based on first order logic.
- They utilize this semantics to provide input to the deductive classifier

- **Machine learning systems –**
- evolve their behavior over time based on experience.
- This may involve reasoning over observed events or example data provided for training purposes - For example, they may use inductive reasoning to generate hypotheses for observed facts.
- Learning systems search for generalized rules or functions that yield results in line with observations and then use these generalizations to control future behavior

- **Case-based reasoning systems** (CBR)
- They provide solutions to problems by analysing similarities to other problems for which known solutions already exist.
- They use analogical reasoning to infer solutions based on case histories.
- commonly used in customer/technical support and call centre scenarios and have applications in industrial manufacture, agriculture, medicine, law and many other areas

- **Procedural reasoning systems** (PRS)
- Uses reasoning techniques to select plans from a procedural knowledge base • Each plan represents a course of action for achievement of a given goal

- The PRS implements a belief-desire-intention model by reasoning over facts ('beliefs') to select appropriate plans ('intentions') for given goals ('desires')
- Typical applications of PRS include management, monitoring and fault detection systems.

DO IT YOURSELF [WARM UP EXERCISE 2]

a. Discuss the concept of conflict resolution and its implementation in predicate logic.

b. Given the following statements and sentences:

Mary is married to James
Lilian is married to John
Shantel, Voke, Lydia and mike are children of Mary Janet
is a child of Lilian

Convert these sentence into: **i** Well-formed formulas (wff).

ii Facts and rules.

c. State the characteristics of Prolog programming language and discuss the elements of a logic language like Prolog.

d. Write a prolog program to correspond to the mathematical function factorial. The factorial of a positive integer N is defined as the product of all the integers from 1 to N inclusive, Example the factorial of 6 will read 120: 1 x 2 x 3 x 4 x 5 x 6 = 720.

e. Using examples explain the difference between Atomic formulas and Compound formulas as used in predicate logic

f. Explain how the prolog compiler will treat each of the following statements.

```
X is X+1.
X1 is X+1.
X1 = X+1.
```

g. Explain how a goal driven system and a data driven system work in relation to an expert system.

h. Write a prolog program to reverse the elements of a list.

i. Find a refutation from the following set of clauses using resolution and factoring techniques.

{P(x, b), P(a, y)} {¬P(x, b), ¬P(c, y)} {¬P(x, d), ¬P(a, y)}

j. Write a prolog program to sum up Even numbers between zero to a number N example given number N as 9 the program gives 8+6+4+2=20.

k. What is first order logic?

l. Explain the meaning of syntax and semantic validity as used in logic programming.

m. Define the following concepts as used in the study of PROLOG: i. Binding Variables ii. Backtracking

iii. Cut function/operator

n. Define and test a predicate which takes two arguments, both numbers, and calculates and outputs the following values:

i. Their average

ii. The square root of their product

a. You are provided with the information lung diseases. Study it and answer the questions that follow:

- **Tuberculosis** is a lung disease whose symptoms are persistant cough, constant fatigue, weight loss, loss of appetite, fever, coughing up blood, night sweats.
- **Pneumonia** is a disease whose symptoms are cough, fever, shaking chills, shortness of breath.
- **Byssinosis** is a disease whose symptoms are chest tightness, cough, wheezing.
- **Pertusis** is a disease whose symptoms are runny nose and mild fever.
- **Pneumoconiosis** is a disease whose symptoms are chronic cough and shortness of breath.

i Write the prolog code to show how the diseases and their respective symptoms will be stored in the knowledge base.

ii Write down a prolog query that will return the Symptoms for Pertusis.

iii Explain how prolog compiler arrives at the solution of the (ii) query above.

REVISION QUESTIONS

a. Write down a rule that satisfies: john likes X:- X is_female, X owns Y, Y is_acat.

b. Study the program below and write down the output for the query indicated.

```
loop(0).
loop(N):-N>0,write('The value is: '),write(N),nl, M
is N-1,iterate(M).
```

?- loop(5).

c. Define and test a predicate which takes two arguments, both numbers, and calculates and outputs the following values: i. Their sum ii. The largest number

d. Explain the requirements of a knowledge representation language.

e. Using examples explain the difference between Atomic formulas and Compound formulas as used in predicate logic

QUESTION 2

a. What is Prepositional Logic and how is it used for Knowledge representation?

b. Differentiate between Syntax and Semantics as used in Logic programming.

c. What is First Order Logic? Does it use quantifiers, if yes, why?

d. Name the various techniques for Knowledge representation.

e. Find a refutation from the following set of clauses using resolution and factoring techniques.

{P(x, b), P(a, y)} {¬P(x, b), ¬P(c, y)} {¬P(x, d), ¬P(a, y)}

f. Prove the following formulas by resolution, showing all steps of the conversion into clauses.

Remember to negate first! $\forall x\,(P \vee Q(x)) \rightarrow (P \vee \forall x\, Q(x))$ $\exists x\, y\,(R(x, y) \rightarrow \forall zw\, R(z, w))$

{Note that P is just a predicate symbol, so in particular, x is not free in P.}

QUESTION 3

a. Discuss any **FOUR** components that define the structure of a prolog program with an example in each case.

b. Explain the operational behaviour of 'CUT' operator in Prolog.

c. Consider the following example:

```
takes(jane, bsc201).
takes(jane, bcs254). takes(chandra,
bit302). takes(chandra, bcs254).
classmates(X, Y) :- takes(X, Z),
takes(Y, Z).
```

i. identify the facts and rules in the program **ii.**
Explain how rules will be evaluated in the program. **iii.**
Write a prolog query that retrieves Y who is a classmate of jane.

iv. Explain the meaning of unification and instantiation using the output of **(iii)** as example.

QUESTION 4

a. What is Knowledge base?

b. Consider prolog program below and explain how unification and instantiation take place after querying.

```
Facts :
   likes(john, jane).
likes(jane, john).

Query :
       ?- likes(john, X).
       Answer: X = jane.
```

c. Differentiate between a well-formed and a not well-formed formula with the help of a suitable example.

d. Write a predicate in Prolog to print the factorial of an integer N. Write down a query that will return the factorial of 6!.

e. Explain the difference between Iteration and Recursion with the help of an example.

f. Write an iterative and recursive program in Prolog to add the elements of a given list of integers.

More Questions

1. Explain the primary focus of Knowledge Representation and Reasoning in the field of Artificial Intelligence.
2. Discuss the role of Knowledge Representation in making complex software easier to define and maintain.
3. Differentiate between declarative, procedural, and meta knowledge, providing examples for each type.

4. How does meta knowledge contribute to handling uncertainty in a knowledge representation system?
5. Discuss the trade-off between expressivity and practicality in designing knowledge representation formalisms.
6. Explain the drawbacks of using First Order Logic (FOL) as a knowledge representation formalism.
7. Describe the role of Knowledge Representation and Reasoning in the Semantic Web.
8. Explain how ontologies contribute to knowledge representation and the challenges they address.
9. Provide an overview of common approaches to handling uncertainty in reasoning systems.
10. Discuss the application scenarios where probabilistic methods like Bayesian inference are particularly useful.
11. Compare and contrast constraint solvers, theorem provers, and logic programs in terms of their applications and characteristics.
12. Explain how machine learning systems differ from rule-based systems in the context of reasoning.
13. Describe the key principles of Case-Based Reasoning (CBR) and provide examples of its application.
14. Discuss the belief-desire-intention model and how it is implemented in Procedural Reasoning Systems (PRS).
15. Explain the role of ontology languages in ontology engineering.
16. Compare frame languages and first-order logic-based languages in the context of knowledge representation.
17. Discuss typical applications of theorem provers, rule engines, and deductive classifiers.
18. Provide examples of real-world scenarios where procedural reasoning systems (PRS) are commonly applied.
19. Explain the evolution of knowledge-based systems, highlighting the shift toward larger and modular knowledge bases.
20. Discuss the motivation behind the rise of ontology engineering and its significance in scaling up knowledge-based technology.