Logical Agents

Sailee Nikhil Panchbhai

Syllabus:

Using predicate logic: Representing simple fact in logic, Representing instant & ISA relationship, Computable functions & predicates, Resolution, Natural deduction. Representing knowledge using rules: Procedural versus declarative knowledge, Logic programming, forward versus backward reasoning, Matching, Control knowledge. First-order logic: Representation Revisited Syntax and Semantics of First-Order Logic, Knowledge Engineering in First-Order Logic Inference in first-order logic, propositional vs. first-order inference, unification & lifts forward chaining, Backward chaining, Resolution

Predicate Logic

- We have already studied how to represent sentences in propositional logic but unfortunately in propositional logic, we can only represent facts that are either true or false.
- Propositional Logic is not sufficient to represent complex sentences.

Some humans are intelligent.

MSD likes Cricket.

} This kind of conversion is not possible using propositional logic due to the existence of key word “Some”

Predicate Logic

- It is another way of knowledge representation in artificial intelligence
- Known as FOPL (First order predicate logic) and FOL (First order logic)
- Extension of propositional logic (connectives: $\Lambda, \mathrm{V}, \sim, \rightarrow, \leftrightarrow$)
- What FOL will follow to represent the things in the real world?
  1. Object: People, numbers, colors
  2. Constant: a proper noun (name of a person, place)
  3. Variable: x, y, a, b, c…….etc
  4. Function: F_name(------)

Fact representation

Example: A is a brother of B

brother_of (A,B)

Predicate Logic:

Predicate:

- the part of a sentence with the verb tells us what the subject is or does. In the sentence—‘He went cycling after returning from school’ the predicate is ‘went cycling after returning from school’
- Relation between two objects

FOL

- A very basic sentence is called an atomic sentence.
- These sentences are formed in FOL by a predicate symbol followed by some parenthesis with the sequence of terms (subjects).

Syntax : Predicate (term1,…….$term_n$)

1. X is a brother of Y

brother_of (X,Y)

brother(X,Y)

2. X is an integer

is_an_integer (X)

integer (X)

FOL

3. Dove is a bird

Is_a_bird (Dove)

Bird (Dove)

Quantifiers

- Defines the scope of the object
- Types of Quantifier:
  1. Universal quantifier (∀)
  2. Existential quantifier (∃)

Quantifiers

- $(\forall)_x$:
  - For each x
  - For all x
  - For every x

- $(\exists)_x$:
  - There exists x
  - For some x
  - For at least one x

FOL

All men drink tea →

X1 drinks tea

Λ

X2 drinks tea

Λ

.

.

.

Xn drinks tea.

Some boys are smart →

X1 is smart

V

X2 is smart

V

.

.

.

Xn is smart.

Note1:

$\forall : \rightarrow$

$\exists : \Lambda$

Note2:

$\forall\, x \forall y = \forall\, y \forall x$

$\exists x \exists y = \exists y \exists \mathrm{x}$

Note3:

$\exists x \forall y \neq \forall\, y \exists \mathrm{x}$

All birds fly

Q S P

Q_{Var} : Subject * Predicate

Connectives

Variable:

X = bird

X is a bird

$\forall x$: is_a_bird(X)

bird (X)→ fly(X)

Some girls are intelligent.

$$\exists x \ : \ \text{girl(X)} \ \Lambda \ \text{intelligent (X)}$$

All men drink tea

$\forall\, x$: man(X)→drink (X,tea)

Some employs are sick.

$\exists x$: employ(X) Λ sick (X)

Every man respects his parent

$$\forall\, x\colon \quad \text{man(X)} \rightarrow \text{respect (X, Parent)}$$

Ravi is a brother of Ram

Brother_of (Ravi,Ram)

Some boys play cricket

$\exists x$: boys(X) Λ Play (X,cricket)

Every student takes at least one course.

$\forall\, x$: students(X)→ $\exists y$ (course(y) Λ takes(x,y))

Not all students like both Mathematics and science

$\neg\forall\ x$: students(X)→likes (X,Mathematics) $\wedge\ likess$(X,Science)

Every student who takes Analysis also takes Geometry

$\forall\, x :$ students(X) Λ takes(X, Analysis) $\longrightarrow$ tkaes (X,Geometry)

No student failed Chemistry but at least one student fail History

$\neg\exists x$: students(X) $\wedge$ $Fail$ (X,Chemistry) $\wedge$ $\exists x$ $student$(X) $\wedge$ $Fail$ (X,History)

Marcus is a man

Man(Marcus)

Marcus was a Pompien

Pompien(Marcus)

All Pompiens were Roman

$\forall\, x$:Pompien(x)→Roman(x)

Every gardener likes Sun

$\forall\, x$:Gardner(x)→likes(x,Sun)

All purple Mushrooms are poisonous

$\forall\, x$:Mushrooms(x) $\wedge\, Purple(X)$ →Poisonus(x)

Everyone is Loyal to Someone

$\forall\, x\ \exists y$: loyal(x,y)

Everyone loves everyone

$\forall\, x \forall\, y$: loves(x,y)

Everyone loves everyone except himself

$\forall\, x \forall\, y$: loves(x,y) $\wedge \neg love(x, x)$

All Romans were either loyal to Ceasar or hated him

$\forall x$: Romans(x) → loyal(x,Ceasar) V hate(x,Ceasar)

Connection between ∀ and ∃

$\forall_x$ Likes (x, Orange)

is equivalent to

$\sim\exists_x \sim$ Likes (x,Orange)

It means that “There does not exist at least one x who does not like an orange” is ultimately equivalent to “All x likes orange”

Representing Instance and isa relationship

- Specific attributes instance and isa play an important role, particularly in a useful form of reasoning called property inheritance.
- The predicates instance and explicitly captured the relationships they used to express, namely class membership and class inclusion.

Instance: Indicates class membership

Isa: indicates class inclusion

Three ways of representing class membership

- First way (Representation)

1. Man(Marcus)
2. Pompeian (Marcus)
3. ∀ x: Pompeian(x) → Roman(x)
4. Ruler (Caesar)
5. ∀ x: Romans(x) → loyalto(x,Ceasar) V hate(x,Ceasar)

- Second way (Use of instance predicate)

1. Instance (Marcus, man)
2. Instance (Marcus, Pompeian)
3. $\forall x$: Instance(x, Pompeian) → Instance(x, Roman)
4. Instance (Caesar, Ruler)
5. $\forall x$: instance(x,Roman) → loyalto(x,Ceasar) V hate(x,Ceasar)

- Third way (use both instance and isa predicate)

1. Instance (Marcus, man)
2. Instance (Marcus, Pompeian)
3. Isa(Pompeian, Roman)
4. Instance (Caesar, Ruler)
5. $\forall\, x$: instance(x,Roman) → loyalto(x,Ceasar) V hate(x,Ceasar)
6. $\forall\, x\, \forall\, y\, \forall\, z : instance(x, y) \wedge isa(y, z) \rightarrow$

Computable functions and Predicates:

- It includes simple facts such as greater than (gt) or less than (lt) relationships

  gt(1,0) lt(0,1)

  gt(2,1) lt(1,2)

  gt(3,2) lt(2,3)

- gt(2+3,1): we first compute the value of the plus function given the arguments 2 and 3 , then send the arguments 5 and 1 to gt.

- Consider the following sets of facts, again involving Marcus:

1. Marcus was a man

   man(Marcus)

2. Marcus was a Pompeian.

   Pompeian(Marcus)

3. Marcus was born in 40 A.D.

   born(Marcus,40)

4. All men are mortal

   $\forall x$: man(x)→Mortal(X)

5. All Pompeians died when Volcano erupted in 79A.D.

   erupted(volcano,79) $\wedge \forall x : [Pompeians(x) \rightarrow died(x, 79)]$

6. No mortal lives longer than 150 years.

$\forall\, x$: $\forall\, t1$: $\forall$ t2: mortal(x) $\wedge\, born(x, t1) \wedge\, gt(t2 - t1,150) \rightarrow died(x, t2)$

7. It is now 1991.

now = 1991

Now suppose we want to answer the question “Is Marcus alive?”

a) Either we can show that Marcus is dead because he was killed by the volcano.

b) He must be dead because he would otherwise be more than 150 years old, which we know is not possible.

8. Alive means not dead.

$\forall x: \forall t: [alive(x,t) \rightarrow \neg \text{dead(x,t)}] \wedge [\neg \text{dead(x,t)} \rightarrow alive(x,t)]$

9. If someone dies, then he is dead at all later times.

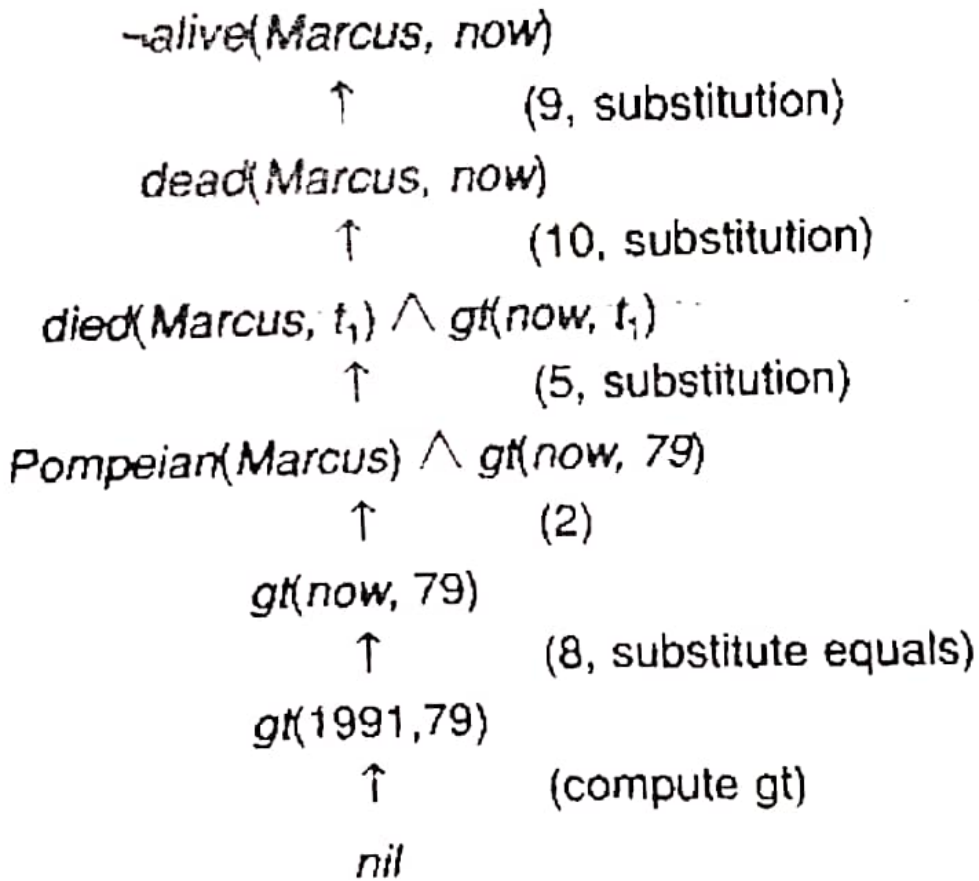
$\forall x: \forall t1: \forall \text{ t2: died(x,t1)} \wedge \text{ gt(t2,t1)} \rightarrow dead(x,t2)$

- Now let’s attempt to answer the question “is Marcus alive?”

By proving $\neg\, alive(x, t)$

A set of facts about Marcus

1. man(Marcus)
2. Pompeian(Marcus)
3. Born(Marcus,40)
4. ∀ x:man(X)→mortal(x)
5. ∀ x:Pompeian(X)→died(x,79)
6. Erupted(volcano,79)
7. ∀ x: ∀ $t1$: ∀ t2: mortal(x) ∧ $born(x, t1) \wedge gt(t2 - t1,150) \rightarrow died(x, t2)$
8. Now =1991
9. ∀ x: ∀ t: $[alive(x, t) \rightarrow \neg$dead(x,t)] ∧[¬dead(x,t)] → $alive(x, t)]$
10. ∀ x: ∀ $t1$: ∀ t2: died(x,t1) ∧ gt(t2,t1) → $dead(x, t2)$


¬alive(Marcus, now)
↑ (9, substitution)
dead(Marcus, now)
↑ (10, substitution)
died(Marcus, t1) ∧ gt(now, t1)
↑ (5, substitution)
Pompeian(Marcus) ∧ gt(now, 79)
↑ (2)
gt(now, 79)
↑ (8, substitute equals)
gt(1991,79)
↑ (compute gt)
nil

Resolution

- Resolution is a theorem-proving technique that proceeds by building refutation proofs, i.e. proofs by contradictions.
- Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements.

Steps for resolution

1. Negate the statement to be proved
2. Conversion of all facts into first-order logic
3. Convert the FOL statement into CNF
4. Draw the resolution graph

Facts:

1. Arya likes all kinds of food
2. Apples and vegetables are food
3. Anything anyone eats and is not killed is food.
4. Ajay eats peanuts and is still alive

Prove: Arya likes peanuts.

1.Negate the statement to be proved

Prove: Arya likes peanuts.

likes(Arya, peanuts)

$\neg\ likes(Arya, peanuts)$

2. Conversion of all facts into first-order logic

1. Arya likes all kinds of food

∀ x:food(X)→likes (Arya,x)

2. Apples and vegetables are food.

$food(Apple) \wedge food(vegetable)$

3. Anything anyone eats and is not killed is food.

$\forall x \forall y : eats(x, y) \wedge \neg$ killed $(x) \rightarrow food(y)$

4. Ajay eats peanuts and still alive

$eats(Ajay, peanuts) \wedge alive(ajay)$

3.Convert the FOL statement into CNF

1. Eliminate '→' and '←→'

$$a \rightarrow b = \neg a \vee b$$

$$a \leftrightarrow b = a \rightarrow b \wedge b \rightarrow a$$

2. Move '¬' inward:

¬(∀ xp) = ∃x¬p

¬(∃xp) = ∀ x ¬p

¬(a ∨ b) = ¬a ∧¬ b

¬(a ∧ b) = ¬a ∨ ¬b

¬¬a = a

3. Rename variable

4. Replace existential quantifier with Skolem constant

$$\exists x\ Rich(x) = Rich\ (G1)$$

5.Drop universal quantifier.

1. ∀ x:food(X)→likes (Arya,x)

   ¬Food(x) ∨ likes (Arya,x)

2. $food(Apple) \wedge food(vegetable)$

3. ∀ x ∀ y :$eats\ (x, y)$ ∧¬ killed (x) → $food\ (y)$

   ¬[eats(x,y) ∧ ¬$killed(x)$] ∨ food (y)

   ¬ eats(x,y) ∨ killed(x) ∨ food(y)

4. $eats(Ajay, peanuts) \wedge alive(ajay)$

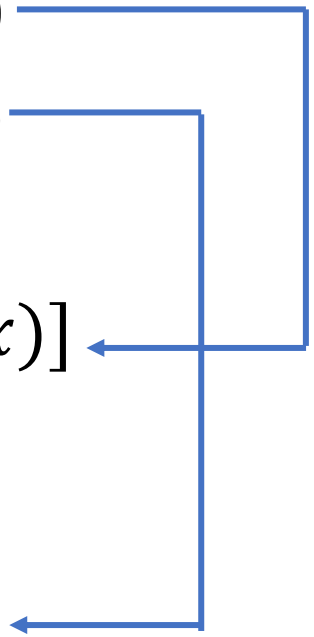
$\forall\, x$: ¬killed (x)→alive(x)

$\forall\, x$: alive(x)→ ¬killed(x)

$\neg\, [\neg killed(x)\ V\ alive(x)]$

$killed(x)\ V\ alive(x)$

¬ alive (x) V ¬killed(x)



4. Resolution graph

CNF(Conjuctive Norml Form):

1. ¬Food(x) V likes (Arya,x)
2. $food(Apple) \wedge food(vegetable)$
3. ¬ eats(x,y) V killed(x) V food(y)
4. $eats(Ajay, peanuts)\ and\ alive(ajay)$
5. $killed(x)\ V\ alive(x)$
6. ¬ alive (x) V ¬killed(x)

Prove: Arya likes peanuts.

$\neg\ likes(Arya, peanuts)$

Procedural versus Declarative knowledge

Procedural Knowledge	Declarative Knowledge
Procedural knowledge means how a particular thing can be accomplished	Declarative knowledge means basic knowledge about something
It is also known as interpretive knowledge	It is also known as descriptive knowledge
Not more popular	More popular
Procedural knowledge can’t be easily communicate	Declarative knowledge can be easily communicate
Procedural knowledge is generally process-oriented in nature	Declarative knowledge is data oriented in nature
In Procedural knowledge debugging and validation is not easy	In Declarative knowledge debugging and validation is easy
Procedural knowledge is less effective in competitive programming	Declarative knowledge is more effective in competitive programming

Forward-chaining	Backward-chaining
Starts with the initial facts.	Starts with some hypothesis or goal.
Asks many questions.	Asks few questions.
Tests all the rules.	Tests some rules.
Slow, because it tests all the rules.	Fast, because it tests fewer rules.
Provides a huge amount of information from just a small amount of data.	Provides a small amount of information from just a small amount of data.
Attempts to infer everything possible from the available information.	Searches only that part of the knowledge base that is relevant to the current problem.
Primarily data-driven	Goal-driven
Uses input; searches rules for answer	Begins with a hypothesis; seeks information until the hypothesis is accepted or rejected.
Top-down reasoning	Bottom-up reasoning
Works forward to find conclusions from facts	Works backward to find facts that support the hypothesis
Tends to be breadth-first	Tends to be depth-first
Suitable for problems that start from data collection, e.g. planning, monitoring, control	Suitable for problems that start from a hypothesis, e.g. diagnosis
Non-focused because it infers all conclusions, may answer unrelated questions	Focused; questions all focused to prove the goal and search as only the part of KB that is related to the problem
Explanation not facilitated	Explanation facilitated
All data is available	Data must be acquired interactively (i.e. on demand)
A small number of initial states but a high number of conclusions	A small number of initial goals and a large number of rules match the facts
Forming a goal is difficult	Easy to form a goal

Logic

- Logic is not concerned with what is true or false, it concerns what follows what. What conclusions follow from a set of premises.

- It can be defined as the study of principles of correct reasoning.

- The main thing we study in logic are principles governing the validity of arguments and checking whether a certain conclusion follows from some given assumption.

Consider:

Nivan likes everyone who likes logic.

Nivan likes logic.

Nivan likes himself.

Is this argument valid? how do you know?

- The logic process takes in some information called Premises and produces some output called Conclusions.

Logic Programming

- PROLOG (Programming in Logic)is one of the most widely used programming languages in artificial intelligence research. As opposed to imperative languages such as C or Java, it is a declarative programming language.
- In the prolog we declare some facts. These facts constitute the knowledge base of the system.
- We can query against the knowledge base. We get output as affirmative if our query is already in the knowledge base or it is implied by Knowledge Base, otherwise, we get output as negative.

- PROLOG facts are expressed in Horn clause (subset of FOL) or simply we can say in FOL
- Every fact ends with a dot (.)

Example:

Friends (Sailee, Ashwini). : Sailee and Ashwini are friends

Singer (Sonu). : Sonu is a singer

Odd_number(5). : 5 is an odd number

Query 1: ?-Singer Sonu.

Output: Yes

Explanation: As our knowledge base contains the above fact, so the output was ‘Yes’, otherwise it would have been ‘No’

Query 1: ?-odd_number (7)

Output: No

Explanation: As our knowledge base does not contain the above fact, so the output was ‘No’

Horn clause

- A Horn Clause is a disjunction with at most one positive literal.

$$\neg\, x1 \;\vee \neg \mathrm{x2} \vee \neg \mathrm{x3} \vee \ldots\ldots\ldots \neg \mathrm{xn} \vee \mathrm{y}$$

Convert the following sentence in to Horn clause.

$$(x1 \;\wedge x2 \wedge \cdots \wedge xn) \rightarrow y$$

Many facts (not all) can be translated into Horn Clause.

For example,

1. $(A \wedge B) \rightarrow C$
2. $(A \vee B) \rightarrow C$
3. $(A \wedge \neg B) \rightarrow C$

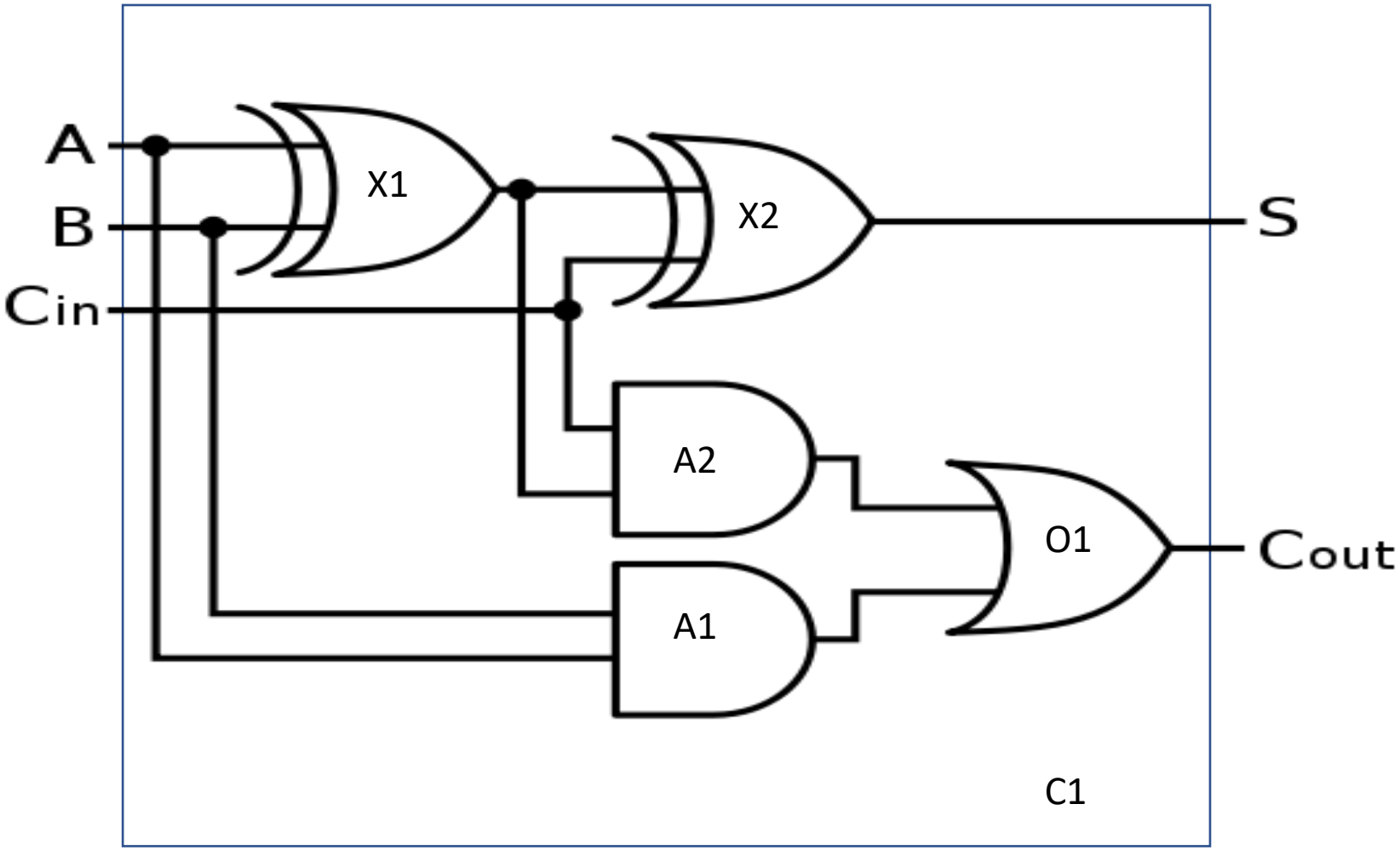
Knowledge Engineering in First-Order Logic:

- The process of constructing a knowledge base in first-order logic is called as knowledge- engineering.
- In knowledge-engineering, someone who investigates a particular domain, learns important concept of that domain, and generates a formal representation of the objects, is known as knowledge engineer.

Knowledge engineering process:

1. Identify the task
2. Assemble the relevant knowledge
3. Decide on vocabulary
4. Encode general knowledge about the domain
5. Encode a description of the problem instance
6. Pose queries to the inference procedure and get answers
7. Debug the knowledge base

1 bit adder:



1. Identify the task

At the first level, examine the functionality of the circuit:

1. Does the circuit add properly?
2. What will be the output of gate A2, if all the inputs are high?

At the second level, we will examine the circuit structure details:

1. Which gate is connected to the first input terminal?
2. Does the circuit have a feedback loop?

2. Assemble the relevant knowledge

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:

- Logic circuits are made up of wires and gates.
- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are three types of gates used: AND, OR, XOR
- All these gates have one output terminal and two input terminals.

3. Decide on vocabulary

- The next step of the process is to select predicates to represent the circuits, terminals, signals, and gates.
- Each gate is represented as an object which is named by a constant, such as **Gate(X1)**.
- Circuits will be identified by a predicate: **Circuit (C1)**.
- For the terminal, we will use the predicate: **Terminal(x)**.
- For gate input, we will use the function **In(1, X1)** for denoting the first input terminal of the gate, and for the output terminal, we will use **Out (1, X1)**.

- The function **Arity(c, i, j)** is used to denote that circuit c has i input, j output.
- The connectivity between gates can be represented by predicate **Connect(Out(1,X1),In(1,X1))**
- The connectivity between gates can be represented by predicate **Connect(Out(1, X1), In(1, X1))**.
- We use a unary predicate **On (t)**, which is true if the signal at a terminal is on.

4. Encode the general knowledge about the domain

1. If two terminals are connected, then they have the same signal:

$$\forall\, t1, t2 : Terminal(t1) \wedge terminal(t2) \wedge connected(t1, t2) \rightarrow signal(t1) = signal(t2)$$

2. Signal at every terminal is either 1 or 0:

$$\forall\, t : Terminal(t) \rightarrow Signal(t) = 1 \ \vee signal(t) = 0$$

3. Connected is commutative

$$\forall\, t1, t2 : connected(t1, t2) \leftrightarrow connected(t2, t1)$$

4. There are three types of gates:

$$\forall\, g : Gate(g) \wedge k = type(g) \rightarrow k = AND \ \vee k = OR \ \vee \mathrm{k} = \mathrm{XOR}$$

5.An AND's gate output is 0 if and only if any of it's input is 0:

$$\forall\, g : Gate(g) \wedge type(g) = AND \ \rightarrow signal(out(1, g) = 0 \ \leftrightarrow$$
$$\ni n\ signal\big(In(n, g)\big) = 0$$

6. An OR gate's output is 1 if and only if any of its input is 1 :

$$\forall\, g : Gate(g) \wedge type(g) = OR \;\rightarrow signal(out(1, g) = 1 \;\leftrightarrow$$
$$\ni\, n\; signal\big(In(n, g)\big) = 1$$

7. An XOR gate's output is 1 if and only if any of its inputs are different :

$$\forall\, g : Gate(g) \wedge type(g) = XOR \;\rightarrow signal(out(1, g) = 1$$
$$\leftrightarrow \;\; signal\big(In(1, g)\big) \;\neq signal(In(2, g))$$

8. A gates have two inputs and one output :

$$\forall\, g : Gate(g) \wedge k = \ type(g) \wedge (k = AND \ \vee k = OR \ \vee k = XOR) \rightarrow Aritty(g, 2,1)$$

9. Gates are circuits.

$$\forall\, g : Gate(g) \ \rightarrow circuit\ (g)$$

5. Encode a description of the problem instance

First , we categorize the circuit and its component gates:

$$circuit(c1) \wedge Arity(c1,3,2)$$
$$\text{Gate}(x1) \wedge Type(x1) = XOR$$
$$\text{Gate}(x2) \wedge Type(x2) = XOR$$
$$\text{Gate}(A1) \wedge Type(A1) = AND$$
$$\text{Gate}(A2) \wedge Type(A2) = AND$$
$$\text{Gate}(O1) \wedge Type(O1) = OR$$

- Now we show the connection between them.

Connected(out(1,x1),In(1,x2))

Connected(out(1,x1),In(2,A2))

Connected(out(1,A2),In(1,O1))

Connected(out(1,A1),In(2,O1))

Connected(out(1,X2),out(S,C1))

Connected(out(1,O1),out(Cout,C1))

Connected(In(A,C1),In(A,X1))

Connected(In(A,C1),In(A,A1))

Connected(In(B,C1),In(B,X1))

Connected(In(B,C1),In(B,A1))

Connected(In(Cin,C1),In(Cin,X2))

Connected(In(Cin,C1),In(1,A2))

6. Pose queries to the inference procedure

- In this step, we will find all the possible sets of values of all the terminals for the adder circuit. The first query will be:
- What should be the combination of input that would generate the first output of circuit C1(sum bit), as 0, and a second output(carryout bit) to be 1?

7. Debug the knowledge base

- Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of the knowledge base.
- In the knowledge base, we may have omitted assertions like $1 \neq 0$

Unification

- Unification is a process of making two different logical atomic expressions identical by finding a substitution.
- Unification depends on the substitution method.

Example of unification:

1. $p(x, f(y))\ and\ p(a, f(g(z))$

Sol: $p(x, f(y)$

$p(a, f(g(z))$

[a | x]

$p(a, f(y)$

$p(a, f(g(z))$

[g(z) | y]

$p(a, f(g(z))$

$p(a, f\big(g(z)\big)$

[a | x , g(z) | y]

2. $Q\big(a, g(x,a), f(y)\big)$ *and* $Q(a, g(f(b),a), x)$

Sol : $Q\big(a, g(x,a), f(y)\big)$

$Q(a, g(f(b),a), x)$

$Q\big(a, g(x,a), f(y)\big)$

$Q(a, g(f(b),a), x)$

$[f(b) \mid x]$

$Q\big(a, g(f(b),a), f(y)\big)$

$Q(a, g(f(b),a), f(b))$

[b | y]

$Q\big(a, g(f(b),a), f(b)\big)$

$Q(a, g(f(b),a), f(b))$

$[f(b) \mid x$, b | y]

Consider $p\big(x, g(x)\big)$

1. $p(z, y\ )$
2. $p\big(z, g(z)\big)$
3. $p\big(socrates, g(socrates)\big)$
4. $p(g(y), z)$
5. $p\big(socrates, f(socrates)\big)$
6. $p(g(y), y\ )$

Consider $p(x, g(x))$

1. $p(z, y)$: $unifies\ with$ [$x|z$, $g(x)|y$]
2. $p(z, g(z))$:unifies with [x|z or z|x]
3. $p(socrates, g(socrates))$: $unifies\ with$ [$socrates$|x]
4. $p(g(y), z)$: unifies with [g(y)|x, g(g(y))|z]
5. $p(socrates, f(socrates))$:
   $does\ not\ unify\ as\ f\ and\ g\ does\ not\ match$
6. $p(g(y), y)$: no substitution works, does not unify

Conditions for Unification

- Predicate symbols must be the same, expressions with different predicate symbols can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm

Unify (L1,L2)

1. If L1 or L2 is a variable or constant, then: