LECTURE 2
LOGIC PROGRAMMING
PROPOSITIONAL LOGIC

Any Logic consists of:

- language (strings of symbols)
- inference rules (how to manipulate symbols)
- method of assigning the meaning for legal string

SYNTAX: set of legal formulas of the Logic in question. (mechanical rules for combining appropriate symbols)

SEMANTICS: the meaning we attach to the legal formula from the Logic in question.

LECTURE 2
LOGIC PROGRAMMING

We start with PROPOSITIONAL LOGIC

↑ legal formulas are propositions (& their combinations)

SYNTAX OF PROPOSITIONAL LOGIC:

{ STEP 1: legal alphabet
{ STEP 2: legal formulas

Example 1.

a) **English language syntax:**

1. Roman alphabet a, b, ..., z, ., :, &c.
2. Set of English words (e.g. look to Oxford dictionary)

b) **Greek language syntax:**

1. Greek alphabet $\alpha, \beta, \ldots$, ., :, etc.
2. Set of Greek words (look to the Greek dictionary). □

LECTURE 2
LOGIC PROGRAMMING

STEP 1.

DEFINITION 1: alphabet for PROPOSITIONAL LOGIC consists of:

1. two symbols:
   a) T (standing for true – semantics)
   b) F (standing for false – semantics)

2. propositional symbols (usually denoted by $p, q, r, \ldots$):

3. logical connectives:
   a) $\sim$ (standing for not)
   b) $\wedge$ (standing for and)
   c) $\vee$ (standing for or)
   d) $\leftarrow$ (needed for clauses)

4. punctuation symbols:
   (improper symbols)
   a) ( ) (brackets)
   b) , (comma needed for clauses)

LECTURE 2
LOGIC PROGRAMMING

Formal objects (legal formulas) for PROPOSITIONAL LOGIC are built by the above alphabet accoroling to:

STEP 2.

DEFINITION 2: Legal formulas for PROPOSITIONAL LOGIC are:

1*. truth symbols T or F

2*. any propositional symbol P, q, r, ...

3. if f_1 and f_2 are **legal** (well-formed) then:

(i) $\sim f_i$ $\quad i=1,2$

(ii) $f_1 \wedge f_2$

(iii) $f_1 \vee f_2$

are legal (well-formed)

* Note that 1. & 2. contain letters from alphabet of Propositional Logic.

LECTURE 2
LOGIC PROGRAMMING

Example 2. Assume we have 3 propositional symbols $\{p, q, r\}$ then:

category 1: T, F.

category 2: p, q, r (whatever each * symbol represents)

category 3: $\sim p, \sim q, \sim r$;

$p \wedge q$, $p \wedge r$, $q \wedge r$,

$q \wedge p$, $r \wedge p$, $r \wedge q$,

$p \vee q$, $r \vee q$, $r \vee p$,

$q \vee p$, $q \vee r$, $p \vee r$,

$\sim p \wedge p$ $\sim q \wedge q$ $\sim r \wedge r$

$\vdots$

$(q \vee p) \wedge p$, $(q \vee r) \wedge q$,

$\vdots$

□

(*) at that point we do not attach the meaning to p, q or r.

LECTURE 2

LOGIC PROGRAMMING

Brackets are needed to use compound formulas linked with logical connectives

So far the legal formulas are formal strings only (with no meaning whatsoever!).

SEMANTICS for PROPOSITIONAL LOGIC

↑

attaches to each legal formula either **true** or **false**.

$$I : f_{\in \Delta} \longrightarrow \{\text{true}, \text{false}\}^{*}$$

f: legal formula

I: ↑ interpretation

* we deal here only with binary logic.

LECTURE 2

LOGIC PROGRAMMING

We need to cover 3 subclasses of legal formulas from DEF. 2 to determine a given **interpretation I:**

For 1:

$$I(T) = \text{true}.$$
$$I(F) = \text{false}.$$

For 2:

$$I(\text{propositional symbol}) \overset{2}{\doteq} \{\text{true}, \text{false}\}$$

we do it according to common sense - whether they are true or false

1 & 2 are interpreted in an atomic fashion. The compound formula:

For 3:

$$I(f) \overset{2}{\doteq} \{\text{true}, \text{false}\}.$$

LECTURE 2
LOGIC PROGRAMMING

(i)

f	$\sim f$
true	false
false	true

truth table

(ii) & (iii)

f_1	f_2	$f_1 \wedge f_2$	$f_1 \vee f_2$
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

truth table

Note: for further notation we may use **0** for **false** & **1** for **true**. Assigning (ii) building (iii) uniquely detemines compound formulas.

LECTURE 2
LOGIC PROGRAMMING

Example 3. $\Delta = \{\underbrace{(p \wedge (\sim q)) \vee s}_{f'''\text{ expressed as in (iii)}}\}$

We define $\Delta_a = \{p, q, s\}$ atomic set of propositional symbols from Δ.

The number of all interpretations for Δ is the same as the number of functions $\Delta_a \longrightarrow \{true, false\}$. $\overline{\overline{\Delta_a}}$ *

From combinatorics $\equiv \overline{\overline{\{true, false\}}}$

$\equiv 2^3 = 8$

$\mathcal{I}$	p	q	s	f
$\mathcal{I}_1$	1	1	1	1
$\mathcal{I}_2$	1	1	0	0
$\mathcal{I}_3$	1	0	1	1
$\mathcal{I}_4$	0	1	1	1
$\mathcal{I}_5$	0	1	0	0
$\mathcal{I}_6$	0	0	1	1
$\mathcal{I}_7$	0	0	0	0
$\mathcal{I}_8$	1	0	0	1

□

* $\overline{\overline{X}}$ – denotes the number of elements in X.

LECTURE 2
LOGIC PROGRAMMING

Example 4. $p \equiv$ "it is cold"
$q \equiv$ "it is snowing"

When we want to say:
"it is cold and it is snowing"
we can express that within PROPOSITIONAL LOGIC framework:

$$\Delta = \{f \equiv p \wedge q\} \quad \underline{\text{SYNTAX}}$$

Now for **SEMANTICS** of f we may have 4 possible interpretations.
One of them is special

$$I(p) = I(q) = \text{true} \Rightarrow I(f) = \text{true}$$

($I(p)$ ↑, $I(q)$ ↑)

if indeed it is now cold & it is snowing.
Such interpretation will be soon defined as **MODEL** for Δ. □

LECTURE 2
LOGIC PROGRAMMING

We introduce also notational shortcuts:

- the logical connectives have a **default precedence** with $\sim$ having the highest & $\vee$ the lowest precedence.

Thus:

$\underbrace{\sim p \wedge q \vee s}_{f_1}$ strictly speaking is not a legal formula

Due to (•):

$\underbrace{(\sim p) \wedge q) \vee s}_{f_2}$ ← this one is, given (•) is a legal one.

Introducing (•) removes bracket cluttering.

f_1 can be treated as colloquialism of f_2.

LECTURE 2

LOGIC PROGRAMMING

In PROLOG we are interested in a certain type of formulas called

A CLAUSE

DEFINITION 3: A **LITERAL** is a propositional symbol p or its negation $\sim p$

$p \leftarrow$ positive literal

$\sim p \leftarrow$ negative literal

DEFINITION 4:
For PROPOSITIONAL LOGIC A **CLAUSE** is a disjunction of literals.

Thus a clause is a formula:

$$(*)\quad A_1 \vee A_2 \vee \ldots \vee A_m \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n,$$

where each A_i $(1 \leq i \leq m)$ & B_j $(1 \leq j \leq n)$ is a positive literal.

LECTURE 2
LOGIC PROGRAMMING

(*) expresses a clause in

disjunctive normal form.
DNF

There is an alternative form for a clause (semanticly equivalent):

$$A_1, A_2, \ldots, A_m \leftarrow B_1, B_2, \ldots, B_n$$

clausal form
CF

Also we introduce:

$\underbrace{A_1, A_2, \ldots, A_m}$

a head of the clause

↑

sum of positive literals

$\underbrace{B_1, B_2, \ldots, B_n}$

a body of the clause

↑

sum of negative literals

LECTURE 2
LOGIC PROGRAMMING

Example 5. a) $\underset{CF}{Q \leftarrow P} \equiv \underset{DNF}{Q \vee \sim P}$

P	Q	Q∨~P	
1	1	1	
1	0	0	← false
0	1	1	
0	0	1	

So $Q \leftarrow P$ is true apart from one case.

b) recalling de Morgan rules: (DML)

$$\sim(B_1 \wedge B_2 \wedge \ldots \wedge B_n) \overset{*}{\equiv} \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n$$

$$\Downarrow$$

$A_1, \ldots, A_m \leftarrow B_1, B_2, \ldots, B_n$ CF

$\equiv$

$A_1 \vee \ldots \vee A_m \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n$ DNF

$\equiv$

$A_1 \vee \ldots \vee A_m \vee \sim(B_1 \wedge B_2 \wedge \ldots \wedge B_n)$ DML

$$\boxed{A_1, A_2, \ldots, A_m \leftarrow (B_1, B_2, \ldots, B_n)}$$

* any model for left-hand side is a model for right-hand side & vice versa.

Thus **commas** in the **head** of clause are standing for **OR**.

Whereas **commas** in the body of clause are standing for **AND**

□.

Note:

1) we first introduce the important notions for PROPOSITIONAL LOGIC

↑ for didactic reasons

⇓

2) later we generalize all to the ultimate PREDICATE LOGIC

↑

more powerful & PROLOG environment.

LECTURE 2
LOGIC PROGRAMMING

Full power of PROLOG is for PREDICATE LOGIC.

Naive example (PROPOSITIONAL LOGIC)

Example 6.

(i) $Q \leftarrow P$
(ii) P

by Modus Ponens $\Rightarrow Q$ is inferred

Can we mimic this on computer?

$$\Delta = \{ Q \leftarrow P \; ; \; P \}$$

data

↑

well formatted as both are clauses*!

* Note P has empty body.

e) >> yes ← PROLOG replies yes and waits for the next query.

f) >> halt. ⊕ RTN ← leaves PROLOG session.

g) > ← return to LINUX Shell

So without invoking Modus Ponens PROLOG replies: Q is "inferrable from initial data".

□

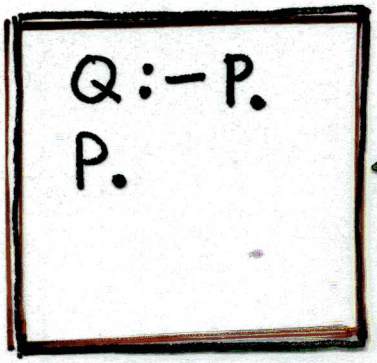
This was a simple example!

One can easily construct complicated set of data & ask PROLOG about the other formulas to be or not inferrable!

LECTURE 2
LOGIC PROGRAMMING

Step 1: we write a program IN PROLOG

example.pl ← any name.pl



←→this represents data Δ

Step 2: we open PROLOG session
by invoking PROLOG interpreter.

a) > pl ⊕RTN Linux shell session

b) >> ← prolog session opens.

c) >>consult('example.pl'). ⊕RTN
← invoking data

ol) >>Q. ⊕RTN
← asking PROLOG whether Q can be inferred.

LECTURE 2
LOGIC PROGRAMMING

For PROPOSITIONAL LOGIC as propositions are variable-free the only query:

is f "inferrable" from Δ?

Not a very important problem.

Soon after introducing PREDICATE LOGIC we shall see more helpful applications!