Part I

Introduction to Logic

Foundations of logic – overview

- Main theme of the course:
  - How to use mathematical logic in
    - programming
    - intelligent web search
- We start with a brief introduction to Logic
  - Propositional Logic:
    - Syntax and semantics
    - The notion of consequence
    - The resolution inference algorithm
    - Bonus: solving various logic puzzles
  - First Order Logic (FOL)
    - Syntax and Model oriented semantics
    - The notion of consequence for FOL
    - The resolution inference algorithm for FOL

Contents

Atomic and compound propositions

- Consider the sentence: *It is raining and I'm staying at home*
- How many propositions (statements) are there in this sentence?
- There are three:
  - two atomic propositions: $A =$ *"It is raining"*, $B =$ *"I'm staying at home"*
  - and the whole sentence is a compound proposition $C = A \wedge B$
  - read the symbol $\wedge$ as "and"
  - C is called a conjunction, A and B are conjuncts
- An atomic proposition is the basic building block of general propositions:
  - it can be assigned a truth value
  - it cannot be broken down to simpler propositions
- Truth values: true and false, often represented by integers 1 and 0
- The term propositional formula (or proposition for short) refers to both atomic and compound propositions

Conjunction

- Knowing the truth values of A and B, can you tell the truth value of $A \wedge B$? Think of $A =$ *"It is raining"*, $B =$ *"I'm staying at home"*

A	B	$A \wedge B$
false	false	false
false	true	false
true	false	false
true	true	true

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

  In brief: $A \wedge B$ is true if and only if (iff) ... both A and B are true
- Is the $\wedge$ operator commutative? I.e. $A \wedge B \stackrel{?}{=} B \wedge A$. Why? Because $0 \wedge 1 = 1 \wedge 0$
- Is $\wedge$ associative? I.e. $(A_1 \wedge A_2) \wedge A_3 \stackrel{?}{=} A_1 \wedge (A_2 \wedge A_3)$. Why? Because both sides are 1 iff each of A_1, A_2, A_3 is 1.
- n-fold conjunction: $C_n = A_1 \wedge \cdots \wedge A_n$. When is $C_n = 1$? If *all* A_is are 1.
- What value should be assigned to an empty conjunction C_0 (C_n for $n = 0$)? Hint: Describe the relationship between C_{n-1} and C_n, use this for $n = 1$ $C_n = C_{n-1} \wedge A_n$, $C_1 = A_1$, hence $A_1 = C_0 \wedge A_1$. This is true iff $C_0 = 1$.

Disjunction and negation

- Another example: *It is not raining or (else) I'm staying at home*
- The two atomic propositions are the same as earlier: $A =$ *"It is raining"*, $B =$ *"I'm staying at home"*
- *"It is not raining"* converts to $\neg A$, where $\neg$ denotes negation, read as "it is not the case that ..."
- The whole sentence can be formalised as $\neg A \vee B$
- Read the symbol $\vee$ as "or"; $A \vee B$ is called a disjunction, A and B are disjuncts
- The truth tables for disjunction and negation (with $0-1$ values only):

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	$\neg A$
0	1
1	0

Implication

- Example: *If it is raining, then drive slower than 100 km/h*
- I obey this sign provided that *If it is raining, then I drive slowly...*
- This is an implication, formally written as $A \rightarrow B$ (A implies B) *the premise:* $A =$ *"It is raining"*, *conclusion:* $B =$ *"I drive slowly ..."*
- When it is not raining, does it matter whether I drive slowly?
- The truth table for implication:

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

- Express implication using disjunction and negation: $A \rightarrow B = \neg A \vee B$
- $A \rightarrow B$ evaluates to 0 iff $A = 1, B = 0$

Equivalence and exclusive or

- Example 1: *I use an umbrella if and only if it is raining*
- This is an equivalence, formally written as $A \leftrightarrow B$ or $A \equiv B$, $A =$ *"I use an umbrella"*, $B =$ *"It is raining"*,
- Example 2: *We either go to movies or have dinner (but not both)*
- This is an exclusive or (XOR), formally written as A *xor* B or $A \oplus B$, $A =$ *"we go to movies"*, $B =$ *"we have a dinner"*
- The truth tables for equivalence and exclusive or:

A	B	$A \equiv B$
0	0	1
0	1	0
1	0	0
1	1	1

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

- Express equivalence using exclusive or, and the other way round: $(A \equiv B) = \neg(A \oplus B)$, $(A \oplus B) = \neg(A \equiv B)$

Normal forms

- A proposition has lots of equivalent formulations:
  $A \to B \equiv \neg A \vee B \equiv \neg(A \wedge \neg B)$
- To design an efficient reasoning algorithm, it makes sense to use one of normal forms (NF), such as:
  - DNF (Disjunctive Normal Form) or CNF (Conjunctive NF)
- Both allow only three operations: $\wedge$, $\vee$, and $\neg$
- In both NFs '$\neg$' can only be used in front of atomic propositions.
- A formula is called a literal if it is either A or $\neg A$, where A is atomic.
- A DNF takes the form $C_1 \vee \ldots \vee C_n$, $n \geq 0$, where each C_i is a conjunction of literals $L_{i1} \wedge \ldots \wedge L_{im_i}$
- A CNF takes the form $D_1 \wedge \ldots \wedge D_n$, $n \geq 0$, where each D_i is a disjunction of literals $L_{i1} \vee \ldots \vee L_{im_i}$
- Transform $A \oplus B$ (exclusive or) to both CNF and DNF formats
- Notice that the DNF can be easily derived from a truth table

Models and tautologies

- Recall two kinds of algebraic formulas from high school:

  $x^2 - 3x + 2 = 0$ equation – true for *some* values of x

  $x^2 - y^2 = (x - y)(x + y)$ identity – true for *all* values of x (*)

- Consider a propositional formula with n atomic propositions, e.g.

$$((A \wedge B) \to C) \quad \equiv \quad (A \to (B \to C))$$

- Here $n = 3$, so there are $2^n = 8$ *valuations* for atomic propositions:
  (A, B, C) can be $(0, 0, 0); (0, 0, 1); (0, 1, 0); \ldots; (1, 1, 0); (1, 1, 1)$
- Each such valuation is called a *model* or a *universe*
- A model satisfies a propositional formula, if the formula is true when the atomic propositions take the 0–1 values specified by the model.
  E.g. the model $(0, 0, 0)$ satisfies the above equivalence
- A formula is called a tautology if all models satisfy the formula
  (cf. the algebraic identity (*) being true for all possible values of x)

Some important tautologies

- Show that this formula is a tautology:

$$((A \wedge B) \to C) \quad \equiv \quad (A \to (B \to C)) \qquad (1)$$

  - Let us find all the models in which the left hand side evaluates to 0:
    There is only one such model $(A, B, C) = (1, 1, 0)$
  - Let us find all the models in which the right hand side evaluates to 0:
    There is only one such model $(A, B, C) = (1, 1, 0)$
  - Hence the above formula is a tautology
- Show that the following formulas are tautologies:

$$\begin{aligned} \neg\neg U &\equiv U \\ \neg(U \wedge V) &\equiv \neg U \vee \neg V \qquad (2) \\ \neg(U \vee V) &\equiv \neg U \wedge \neg V \qquad (3) \end{aligned}$$

  (2) and (3) are called De Morgan's laws.
- Hint: use case-based reasoning for proving formulas (2) and (3):
  1. Select an arbitrary atomic proposition in the formula, say U
  2. Show that the formula to be proven holds for both $U = 0$ and $U = 1$

Contents

An automated inference system: resolution

- The *first order resolution* inference algorithm was devised by Alan Robinson around 1964
- We now introduce resolution for propositional logic
- Resolution uses CNF, *conjunctive normal form* (recall):
  - a CNF is a conjunction of *clauses*: $Cl_1 \wedge \ldots \wedge Cl_n$
  - a clause is a disjunction of *literals*: $L_1 \vee \ldots \vee L_k$
  - a literal is either A or $\neg A$, where A is an atomic proposition

Translating propositions to clausal form

- Steps needed to transform an arbitrary formula to CNF:
  1. replace all connectives by equivalents using only $\neg, \wedge, \vee$
  2. move negations inside using De Morgan Laws
  3. apply distributivity (repeatedly, if needed) to eliminate $\wedge$ s inside $\vee$ s: transform $U \vee (V \wedge W)$ to $(U \vee V) \wedge (U \vee W)$
  4. transform $\wedge$ and $\vee$ operators to sets, elminating duplicates

  The result is thus a set of sets, e.g. $\{\{A, B\}, \{B, C\}\} \equiv (A \vee B) \wedge (B \vee C)$ ("Outer" set elements are conjuncts, "inner" set elements are disjuncts)
- Simplified notation (used in first Prolog versions)
  - a literal is written as a signed atomic proposition, e.g. `-A`, `+B` (for $\neg$`A`, `B`)
  - a clause is written as a sequence of literals followed by a full stop, e.g. $\neg A \vee \neg B \vee D$ written as `-A -B +D.`
- Example: transform $((A \wedge B) \rightarrow D) \wedge (C \rightarrow (A \wedge B))$ to to clausal form
  The CNF form: $(\neg A \vee \neg B \vee D) \wedge (\neg C \vee A) \wedge (\neg C \vee B)$
  The CNF in set notation: $\{\{\neg A, \neg B, D\}, \{\neg C, A\}, \{\neg C, B\}\}$
  The CNF in simplified notation: `-A -B +D. -C +A. -C +B.`

The resolution inference rule – introduction

- Consider these two clauses: `+A -B -C.` (1)
  `+A +D +B.` (2)
- Literal # 2 in clause (1) is `-B`, while literal # 3 in clause (2) is `+B`. These literals are *opposite*, i.e. one is the negation of the other.
- Given two clauses containing opposite literals, the resolution rule infers a new clause, called the resolvent, containing the union of all literals of the two clauses, except the two opposite literals.
- In the example the resolvent clause is `+A -C +D.` (3)
  Note that there is only one `+A` as `A` $\vee$ `A` = `A`.
- Resolution is sound, i.e. (3) is implied by (1) and (2). This is due to the *resolution principle*:

$$\underbrace{(\neg U \vee V)}_{\text{(i)}} \wedge \underbrace{(U \vee W)}_{\text{(ii)}} \rightarrow (V \vee W) \quad (4)$$

- Proof: Assume the LHS is true, so both (i) and (ii) are true.
  - If U is true V has to be true, for disjunction (i) to be true.
  - If U is false W has to be true, for disjunction (ii) to be true.

  In either case the RHS is true.

The resolution inference rule – full definition (ADVANCED)

- Input: two clauses $C = \mathtt{L}_1\ \mathtt{L}_2\ \ldots\ \mathtt{L}_n.$
  $D = \mathtt{M}_1\ \mathtt{M}_2\ \ldots\ \mathtt{M}_k.$

  where $\mathtt{L}_i$ = `+X` and $\mathtt{M}_j$ = `-X`, or $\mathtt{L}_i$ = `-X` and $\mathtt{M}_j$ = `+X`.
- Let $C' = C \setminus \{\mathtt{L}_i\}$, $D' = D \setminus \{\mathtt{M}_j\}$, where $\setminus$ denotes set difference.
  (The set difference $S_1 \setminus S_2$ is obtained by removing all elements of S_2 – if present – from S_1)
  Thus $C' = \mathtt{L}_1\ \ldots\ \mathtt{L}_{i-1}\ \mathtt{L}_{i+1}\ \ldots\ \mathtt{L}_n.$
  $D' = \mathtt{M}_1\ \ldots\ \mathtt{M}_{j-1}\ \mathtt{M}_{j+1}\ \ldots\ \mathtt{M}_k.$
- Resolution of C and D yields the clause $E = C' \cup D'$ (meaning $C' \vee D'$), called the $resolvent_{ij}(C, D)$, or simply $resolvent(C, D)$;
  $E = \mathtt{L}_1\ \ldots\ \mathtt{L}_{i-1}\ \mathtt{L}_{i+1}\ \ldots\ \mathtt{L}_n\ \mathtt{M}_1\ \ldots\ \mathtt{M}_{j-1}\ \mathtt{M}_{j+1}\ \ldots\ \mathtt{M}_k.$
  (with duplicates removed)
- Note that only a single pair of opposite literals is removed by the resolution step!

The resolution rule – remarks

- Informally: the resolution rule can be interpreted as viewing the clauses as arithmetic formulas, to be summed up and removing *exactly one* pair of "summands" `+X -X`
  - Example: *resolvent*(`+A-B-C`, `+B+D`) = `+A-C+D`
  - Remark: this analogy does not work, if there is a literal which occurs in both clauses,
    e.g. *resolvent*(`+A-B-C`, `+B+D+A`) = `+A-C+D` (only one `+A` is kept)
- The case of having two or more "summands" with opposite signs also breaks the analogy
  - Here only one pair of such summands is removed
  - Example: $resolvent_{21}$(`+A-B-C`, `+B+D+C`) = `+A-C+D+C` = 1 (true), or $resolvent_{33}$(`+A-B-C`, `+B+D+C`) = `+A-B+B+D` = 1
  - Thus resolution does not produce a meaningful clause in this case

Example: solving an inspector Craig puzzle using resolution

- The puzzle below is cited from "What Is The Name Of This Book?" by Raymond M. Smullyan, chapter "From the cases of Inspector Craig"
- Puzzles in this chapter involve suspects of a crime, named A, B, etc. Some of them are guilty, some innocent.
- Example:
  An enormous amount of loot had been stolen from a store. The criminal (or criminals) took the heist away in a car. Three well-known criminals A, B, C were brought to Scotland Yard for questioning. The following facts were ascertained:
  1. No one other than A, B, C was involved in the robbery.
  2. C never works without A (and possibly others) as an accomplice.
  3. B does not know how to drive.

  Is A innocent or guilty?

Inspector Craig puzzle – solution

- Let's recall the facts
  1. No one other than A, B, C was involved in the robbery.
  2. C never works without A (and possibly others) as an accomplice.
  3. B does not know how to drive.
- Transform each statement into a formula involving the letters A, B, C as atomic propositions. Proposition A stands for "A is guilty", etc.
  1. A is guilty or B is guilty or C is guilty: $A \vee B \vee C$
  2. If C is guilty then A is guilty: $C \to A$
  3. It cannot be the case that only B is guilty: $B \to (A \vee C)$
- Transform each propositional formula into conjunctive normal form (CNF), then show the clauses in simplified form:

	Original formula	CNF	Simplified clausal form
1	$A \vee B \vee C$	$A \vee B \vee C$	`+A +B +C.`
2	$C \to A$	$\neg C \vee A$	`-C +A.`
3	$B \to (A \vee C)$	$\neg B \vee A \vee C$	`-B +A +C.`

(Note that in general a single formula can give rise to multiple clauses.)

Inspector Craig puzzle – resolution proof

- Collect the clauses, give each a reference number and perform a resolution proof:

```
(1)    +A +B +C.          Only A, B, C was involved in the robbery.
(2)    -C +A.             C never works without A as an accomplice.
(3)    -B +A +C.          B does not know how to drive.

                          resolve (1) lit 2 with (3) lit 1 resulting in (4)
(4)    +A +C.             resolve (4) lit 2 with (2) lit 1 resulting in (5)
(5)    +A.
```

- We deduced that A is true, so the solution of the puzzle is: A is guilty
- Notice that `+A` occurs in each of the above clauses, hence each of `(1)`–`(4)` follows from `(5)`
- This, together with the fact that `(5)` follows from the input clauses `(1)`–`(3)`, means that `(5)` is equivalent to the set of input clauses
- Hence the statements of the puzzle impose no restrictions on propositions B and C
  (either can be guilty or innocent – all 4 combinations allowed)

Removing trivial consequences

Consider this set of clauses: $CS = \{$ `-B+C+D`, `+A+C`, `-A-B`, `+A-B+C` $\}$

- Find a clause in CS that is a consequence of another clause in CS.
- Hint: of these formulas, which implies which other? $U \vee V$, U, V?
  (If we know $U \vee V$ is true, can U be false?) Yes, it can.
  (If we know U is true, can $U \vee V$ be false?) No
- Hence U implies $U \vee V$, and similarly V implies $U \vee V$
- Viewing clauses as sets, if $C \subseteq D$, then $C \rightarrow D$ ("subset" $\rightarrow$ "whole set")
- `+A+C` $\rightarrow$ `+A-B+C`, so `+A-B+C` is a trivial consequence of `+A+C`

Trivial consequences

- A clause $C \vee D$ ($D \neq$ empty) is said to be a trivial consequence of C
- Is it of interest to obtain the set of all consequences of CS?
- No, we get marred by trivial consequences, e.g. `-A-B-C`, `-A-B+C`, ...
- It makes more sense to construct a maximal set of *non-trivial* consequences, i.e. a set MCS which contains all consequences of CS, except those that are a trivial consequence of a clause already in MCS
- Removing a trivial consequence is valid because $(C \wedge (C \vee D)) \equiv C$

Maximal set of non-trivial consequences (ADVANCED)

For the mathematically minded, here is a precise definition of the *maximal set of non-trivial consequences*

- For a set of clauses CS, its maximal set of consequences is MCS iff:
  - *each clause in MCS is a consequence of CS:*
    for each $C \in MCS$, $CS \rightarrow C$
  - *there are no trivial consequences in MCS:*
    for each $C_1, C_2 \in MCS$, C_2 is not a trivial consequence of C_1
  - *MCS contains all non-trivial consequences:*
    for each clause C such that $CS \rightarrow C$ holds, either $C \in MCS$ holds, or else C is a trivial consequence of a $C' \in MCS$.

Constructing MCS – continuing the example

- The set of input clauses:

```
(1) -B+C+D
(2) +A+C
(3) -A-B
(4) +A-B+C
```

- Remove `(4)`, as it is implied by `(2)`
- Resolve `(2)` with `(3)` producing a new clause:

```
(5) -B+C
```

- Remove `(1)`, as it is implied by `(5)`
- As no removal or resolution step can be applied, exit with the following maximal set of (non-trivial) consequences:

```
(2) +A+C
(3) -A-B
(5) -B+C
```


A saturation algorithm for obtaining MCS (ADVANCED)

Given a set of clauses CS_0, you can obtain its maximal set of consequences by performing the following algorithm:

1. set CS to CS_0
2. (exit if inconsistency is detected)
   if CS contains an empty clause, then exit reporting CS_0 is inconsistent
3. (remove a trivial consequence)
   if there are $C_1, C_2 \in CS$ such that C_2 is a trivial consequence of C_1, then remove C_2 from CS, and repeat step 3
4. (perform a meaningful resolution step)
   if there are $C_1, C_2 \in CS$ such that C_1 resolved with C_2 yields C_3 where $C_3 \neq \mathit{true}$ and $C_3 \notin CS$, then add C_3 to CS, and continue at step 3
5. (exit when saturated)
   as the conditions of both steps 3 and 4 failed, exit with $MCS = CS$

Finding a single consequence using an indirect proof

- For large sets of formulas finding all consequences is not viable
- Recall the Inspector Craig puzzle discussed earlier:

```
(1)    +A +B +C.        Only A, B, C was involved in the robbery.
(2)    -C +A.           C never works without A as an accomplice.
(3)    -B +A +C.        B does not know how to drive.
```

- To prove indirectly that (1)–(3) implies A, add ¬ A to the set of clauses:

```
(4)    -A.        ... and perform resolutions, adding resolvents to the set
                  (4)/1 rw (1)/1 (cl. (4) lit. 1 resolved with cl. (1) lit. 1) ⇒ (5)
(5)    +B +C.     (5)/1 rw (3)/1 ⇒ (6)
(6)    +A +C.     (6)/1 rw (4)/1 ⇒ (7)
(7)    +C.        (7)/1 rw (2)/1 ⇒ (8)
(8)    +A.        (8)/1 rw (4)/1 ⇒ (9)
(9)    □          This denotes an empty disjunction ≡ false
```

- Adding ¬A to (1)–(3) leads to contradiction, so {(1),(2),(3)} implies A
- This indirect proof is focused on proving the given statement

  Notice that the above proof is quite mechanical:
  - the first input clause is the result of the previous resolution step
  - we always resolve on the first literal of the first input clause

Inspector Craig puzzle – further proof attempts (ADVANCED)

```
(1)    +A +B +C.
(2)    -C +A.
(3)    -B +A +C.
```

- We now try to prove indirectly that ¬C follows from (1)–(3), by adding C:

```
(4)    +C.                          implied clauses removed: (1), (3)
(5)    +A.       (4)/1 rw (2)/1     implied clauses removed: (2)
```

- The set {(4),(5)} is saturated, hence {(1)–(3)} does not imply ¬ C
- Let's now try to prove that C follows from (1)-(3), by adding ¬ C:

```
(6)    -C.                          implied clauses removed: (2)
(7)    +A +B.    (6)/1 rw (1)/3     implied clauses removed: (1)
(8)    -B +A.    (6)/1 rw (3)/3     implied clauses removed: (3)
(9)    +A.       (7)/2 rw (8)/1     implied clauses removed: (7), (8)
```

- The set {(6),(9)} is saturated, hence {(1)–(3)} does not imply C
- We conclude that neither C nor its negation can be deduced from (1)–(3)
- (However, the first unsuccessful proof shows that if C is true, so is A, while the second proof demonstrates that if ¬C is true, A is true again, so A has to be true. :-)

More puzzles: knights and knaves

- The puzzle below is also cited from the same Raymond M. Smullyan book. It is about an island in which certain inhabitants called "knights" always tell the truth, and others called "knaves" always lie. It is assumed that every inhabitant of the island is either a knight or a knave.
- Puzzle 1: There are two people, A and B. Suppose A says: "Either I am a knave or B is a knight." What are A and B?
- Translate this puzzle to a prop. formula and solve it using resolution
- Here is the syntax of a "controlled English" format for such puzzles:

  ⟨person⟩ ::= knight | knave | A | B | C . . .

  ⟨statement⟩:= ⟨person⟩ = ⟨person⟩ |
  ⟨person⟩ says ⟨statement⟩ |
  not ⟨statement⟩ |
  ⟨statement⟩ and ⟨statement⟩ |
  ⟨statement⟩ or ⟨statement⟩ |
  (⟨statement⟩)

  A puzzle in this format can be fed to a Prolog program which solves it.
- Can you convert Puzzle 1 to the controlled English format?

Further knights and knaves puzzles

- Puzzle 2: There are two people, A and B. A makes the following statement: "If I am a knight, so is B." What are A and B?
- Puzzle 3: There are two people, A and B. A makes the following statement: "At least one of us is a knave." What are A and B?
- Puzzle 4: Suppose A says, "I am a knave, but B isn't." What are A and B?
- Puzzle 5: We now have three people, A, B, C. Two people are said to be of the same type if they are both knights or both knaves. A and B make the following statements:
  A: B is a knave.
  B: A and C are of the same type.
  What is C?

Summary of Propositional Logic

We had a quick overview of Propositional Logic

- Connectives $\wedge, \vee, \neg, \rightarrow, \equiv, \oplus$
- Truth tables for each connective
- Some simple "theorems", e.g. De Morgan's laws
- The notions of semantic and syntactic consequence
- Resolution as a proof system for Propositional Logic
- Solving "knights and knaves" puzzles using propositional logic

Contents

New features of First Order Logic (FOL) wrt. Propositional Logic

- Propositions have a structure: a predicate name followed by a list of arguments in parentheses.
  - English: *"Nick has a parent Mary"*
  - FOL: $has\_parent(Nick, Mary)$
- Variable names and quantifiers
  - English: *"All children of Mary are happy"*
  - FOL: $\forall z.(has\_child(Mary, z) \rightarrow is\_happy(z))$
    Read: for all z, if *Mary* has a child z, then z is happy.
- Equality is available as a special "built-in" predicate
  - English: *"Everyone has at most one mother"*
  - FOL: $\forall x.\forall y.\forall z.(has\_mother(x, y) \wedge has\_mother(x, z) \rightarrow y = z)$
    Read: for all x, y, z, if y and z are both mothers of x, then $y = z$.
- Function expressions, e.g. $mother(Nick)$, meaning *mother of Nick*
  - English: *"A parent is either a father or a mother"*
  - FOL: $\forall x.\forall y.(has\_parent(x, y) \leftrightarrow (y = mother(x) \vee y = father(x)))$
    Read: for all x, y: x has a parent y if and only if
    y is either the mother of x, or y is the father of x.

Representing general knowledge in FOL

- Can you deduce that Z follows from A and B in propositional logic?
  $A =$ `"Nick has a parent Mary"`, $B =$ `"Mary has a parent Paul"`,
  $Z =$ `"Nick has a grandparent Paul"`.
- No, propositions are atomic objects, with no internal structure
- In FOL, we can express the above facts in a structured form:
  A = *has_parent(Nick,Mary)*, B = *has_parent(Mary,Paul)*,
  Z = *has_grandparent(Nick,Paul)*
- To be able to deduce Z from A and B, we have to formalize the general knowledge that parents of parents are grandparents:
  $C = \forall x.\forall z.(\exists y.(has\_parent(x, y) \wedge has\_parent(y, z)) \rightarrow has\_grandparent(x, z))$
  Read: for all x, z,
  if there exists a y such that x has y as a parent, and y has z as a parent
  then x has z as a grandparent.
  (Note that implication $\rightarrow$ could be replaced by an equivalence $\leftrightarrow$)
- In FOL, given A, B and C, one can deduce Z

Another example

- Consider an island inhabited by at least one person
  - Some people (possibly none) are optimistic.
  - A person may have another person as a friend. There is no restriction on the number of friends a person may have, this could be 0, 1, or more. Also, friendship may not be mutual.
- We know the following facts
  - (a) If someone has a non-optimistic friend, then they are optimistic.
  - (b) There is at least one person, who has a friend.
- Try convincing yourself that the following statement must hold:
  - (c) There is an optimistic person on the island.
- Express statements (a), (b) and (c) in FOL using these predicates:
  - $hasF(x, y)$: x has y as their friend
  - $opt(x)$: x is optimistic
- As an example, here is another statement and its FOL formulation. Each optimistic person has a friend: $\forall x.(opt(x) \rightarrow \exists y.hasF(x, y))$

Reasoning in First Order Logic

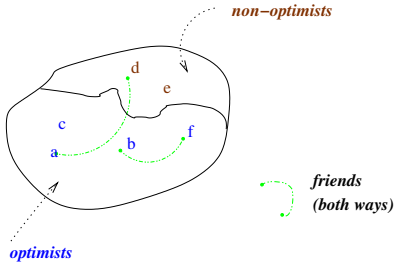
- **Recall:** (a) Someone having a non-optimistic friend is optimistic.
  (b) There is at least one person, who has a friend.
  (c) There is an optimistic person on the island.
- If (a) and (b) are true, can you argue that (c) has to be true?
  - (b) states that there is a person (say `p1`) who has a friend (say `p2`)
  - Do case-based reasoning: `p2` is either optimistic or not
    - Case 1: `p2` is optimistic. This implies that (c) is true
    - Case 2: `p2` is not optimistic. As `p1` has (the non-optimistic) `p2` as a friend, because of (a), `p1` is optimistic. Thus (c) is true again.
  - As both cases lead to (c) being true, we can conclude that whenever (a) and (b) hold on an island, (c) is bound to hold on this island.
- Here (c) is said to be a semantic consequence of $\{(a), (b)\}$:

$$\{(a), (b)\} \models (c)$$

- This is in contrast with syntactic consequence (to be discussed later), which builds on symbolic transformations, such as *modus ponens*: if $U \rightarrow V$ and U are true, one can conclude that V is true, as well.

First Order Logic – Proving a consequence (cont'd)

- The proof on the previous slide works for any island (math-speak: model), e.g.



- A model for this example can be described by
  - a set Δ containing the inhabitants of the island, e.g. $\{a, b, c, d, e, f\}$
  - the interpretation of the 1-argument predicate `opt` $\subseteq \Delta$, i.e. the subset of Δ containing the optimists: $\{a, b, c, f\}$
  - the interpretation of the 2-argument predicate `hasF` $\subseteq \Delta \times \Delta$, i.e. the set of pairs in Δ that are in `hasF` relation: $\{\langle a, d \rangle, \langle d, a \rangle, \langle b, f \rangle, \langle f, b \rangle\}$
- A model has all information needed to check whether a formula is true

First Order Logic – an overview

First Order Logic (FOL)

- Propositional Logic is a special case of FOL where all predicate symbols have 0 arguments (and hence function symbols, variables and quantifiers make no sense)
- Views of logic
  - Syntax (What are the well-formed statements?)
  - Proofs (How can one obtain true statements?)
  - Semantics (What is the meaning of statements and their components?)

Contents

First Order Logic – Syntax

Building blocks of FOL

- Symbols:
  - logical symbols: propositional connectives $\vee$, $\neg$, ...; quantifiers $\forall$ $\exists$, punctuation etc.– these have a *fixed meaning*
  - non-logical symbols such as $hasF$ – these have *arbitrary meaning*

  An analogy with programming languages:
  logical symbols – keywords, non-logical symbols – identifiers
- Terms represent individual objects in our universe, e.g. if $f(x)$ and $m(x)$ denote the father and the mother of x, and $s()$ denotes an individual named Susan, then $m(f(s()))$ refers to Susan's father's mother, i.e. the paternal grandmother of Susan
- Formulas state truths, e.g. $hasF(m(f(s())),m(s()))$ – meaning Susan's paternal grandmother has Susan's mother as a friend.

The alphabet of FOL – the symbols used in formulas

- logical symbols
  - punctuation symbols: (,) .
  - logic connectives:
    $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation),
    $\exists$ (existential quantifier symbol – "exists such ... that ..."),
    $\forall$ (universal quantifier symbol – "for all ... holds that ..."),
    $=$ (equality predicate)
  - variable symbols: $x_1, \ldots, x_i, \ldots$
- non-logical symbols (cf. identifiers in programming languages)
  - function symbols: $f, g, h, \ldots$, (including the special case of) constant (nullary function) symbols: $a, b, c, \ldots$
  - predicate symbols: $p, q, r, \ldots$
  - each function and predicate symbol has a fixed arity (# of args) ≥ 0
- a signature (cf. declaring vars in a program) specifies a set of function and predicate symbols, together with their arities, e.g.
  functions: $f/1$ ($f(x)$ denotes the father of x), $m/1$ ("mother of"),
  predicates: $hasF/2, opt/1$

Syntax of FOL, computer scientists style

⟨ term ⟩	::=	⟨ var symbol ⟩	|
		⟨ function symbol ⟩(⟨ arglist ⟩)	
⟨ arglist ⟩	::=		| % *empty*
		⟨ term ⟩, ...	% *comma sep. list*
⟨ atomic formula ⟩	::=	⟨ pred symbol ⟩(⟨ arglist ⟩)	|
		⟨ term ⟩ = ⟨ term ⟩	
⟨ formula ⟩	::=	⟨ atomic formula ⟩	|
		(¬ ⟨ formula ⟩)	|
		(⟨ formula ⟩ ∧ ⟨ formula ⟩)	|
		(⟨ formula ⟩ ∨ ⟨ formula ⟩)	|
		∃ ⟨ var symbol ⟩ . (⟨ formula ⟩)	|
		∀ ⟨ var symbol ⟩ . (⟨ formula ⟩)	

Mathematicians often

- insert/delete parentheses and/or dots (in quantified formulas)
- omit empty function arguments (), e.g. allow s as a shorthand for $s()$
- use multiple vars after a single quantifier, e.g. $\forall x, y.(\ldots) \equiv \forall x.(\forall y.(\ldots))$

Syntax of FOL, mathematician style (ADVANCED)

- A term is a text (a sequence of symbols) to name an object of the universe of discourse
  - A variable symbol is a term
  - If $t_1, \ldots, t_n$ are terms and f is a function symbol of arity n, then $f(t_1, \ldots, t_n)$ is a term
  - A term of FOL is obtained by applying the above two rules a finite number of times
- A well formed FOL formula (wff) is a text describing a statement
  - If $t_1, \ldots, t_n$ are terms and p is a predicate symbol of arity n, then $p(t_1, \ldots, t_n)$ is an atomic formula
  - If t_1 and t_2 are terms, then $t_1 = t_2$ is also an atomic formula.
  - If α and β are wffs, x is a variable symbol, then $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\exists x.\alpha)$, $(\forall x.\alpha)$ are wffs, too.
  - A well formed formula is obtained by applying the above rules a finite number of times

Syntax of FOL, remarks

- Abbreviations – adding further propositional ops, as "syntactic sugar":
  - $(\alpha \to \beta)$ is an abbreviation of: $(\neg\alpha \vee \beta)$
  - $(\alpha \equiv \beta)$ is an abbreviation of: $((\alpha \to \beta) \wedge (\beta \to \alpha))$
  - note that formulas $(\alpha \vee \beta)$ and $(\exists x.\alpha)$ could also have been defined as abbreviations, using De Morgan's laws (extended to quantifiers):
    - $(\alpha \vee \beta) \equiv \neg(\neg\alpha \wedge \neg\beta)$
    - $(\exists x.\alpha) \equiv \neg(\forall x.\neg\alpha)$
- The scope of variables
  - An occurrence of variable x is bound if it appears inside a formula $\exists x.\alpha$ or $\forall x.\alpha$
  - A variable occurrence x is free if it is not bound
- A formula is a sentence (also called a closed formula) if it contains bound variables only

Some further practice

- Formalize in FOL the statements below, using the signature:
  function symbols: *f*/1 and *m*/1 (for father and mother), *s*/0 for Susan;
  predicate symbols *hasF*/2 (has friend), and *opt*/1 (optimist).
  1. Someone is an optimist. (recall)
  2. Everyone is an optimist.
  3. Everyone has a friend.
  4. There is someone who is befriended with their father's mother.
  5. Someone is not an optimist.
  6. Everyone is a friend of themselves.
  7. If x's father or mother is an optimist, so is x, for any x
  8. If x has a non-optimist friend, then x is an optimist, for any x. (recall)
  9. Anyone whose all friends are optimists is bound to have a friend.
  10. Susan is an optimist.
  11. Susan's maternal grandmother has Susan's paternal grandmother as a friend.
- Try finding subsets of the above FOL sentences so that another sentence above is a consequence of the given subset

Contents

Clauses in First Order Logic

- From now on we assume that there are no equality literals in the clauses (these can be handled using the *paramodulation* technique, not discussed in this course).
- A FOL *clause* is
  - a set of literals (disjuncts),
  - each being a plain or negated *atomic* formula,
  - with all variables universally quantified.
- An example: one's female parent is their mother.
  `-hasParent(x,y) -female(y) +hasMother(x,y).`
  $\equiv \forall x, y.((\texttt{hasParent}(x, y) \wedge \texttt{female}(y)) \rightarrow \texttt{hasMother}(x, y))$
- An arbitrary FOL statement can be transformed to a set of clauses by:
  1. doing propositional transformations
     - expressing $\rightarrow$, $\equiv$ etc, using $\neg$, $\wedge$, and $\vee$
     - bringing $\neg$ inside $\wedge$ and $\vee$ (to appear in front of atomic formulas)
  2. bringing quantifiers to the front of the formula
  3. converting to CNF
  4. getting rid of $\exists$ quantifiers by introducing so called Skolem functions (not relevant in Logic Programming, not discussed further)

A sample transformation to CNF

- Example: if x has a non-optimist friend, then x is an optimist
- FOL formula: $\forall x.(\exists y.(hasF(x, y) \wedge \neg opt(y)) \rightarrow opt(x))$
- Eliminate implication ($U \rightarrow V \equiv \neg U \vee V$):
  $\forall x.(\neg(\exists y.(hasF(x, y) \wedge \neg opt(y))) \vee opt(x))$
- Bring negation inside
  (use $\neg\exists u.W \equiv \forall u.\neg W$, and standard De Morgan rules):
  $\forall x.(\forall y.(\neg hasF(x, y) \vee opt(y)) \vee opt(x))$
- Bring $\forall, \exists$ outside $\forall x.(\forall y.(\varphi_1(x, y)) \ldots \varphi_2(x)) \equiv \forall x, y.(\varphi_1(x, y) \ldots \varphi_2(x))$:
  $\forall x, y.(\neg hasF(x, y) \vee opt(y) \vee opt(x))$
- Transform to Conjunctive Normal Form (CNF):
  $\neg hasF(x, y) \vee opt(y) \vee opt(x)$
- Transform to Simplified CNF: `-hasF(X,Y) +opt(Y) +opt(X).`
- In the simplified CNF format we use the convention that capitalized identifiers denote variables (as is the case for Prolog)

How to read the clausal form?

- A general clause: $-A_1 \ \ldots \ -A_m \ + B_1 \ldots + B_n.$, $m \geq 0, n \geq 0$
- The simplest readout (no $\neg$): the *conjunction* of negative literals implies the *disjunction* of positive literals: $(A_1 \wedge \ldots \wedge A_m) \rightarrow (B_1 \vee \ldots \vee B_n)$
- Example: `-hasF(X,Y) +opt(Y) +opt(X).` can be read as

English	*FOL* (with implicit $\forall$ quantifiers)
One of a pair of friends has to be *opt*.	$hasF(X, Y) \rightarrow opt(Y) \vee opt(X)$
Those with a non-*opt* friend are *opt*.	$hasF(X, Y) \wedge \neg opt(Y) \rightarrow opt(X)$
A friend of a non-*opt* is an *opt*.	$hasF(X, Y) \wedge \neg opt(X) \rightarrow opt(Y)$
Two non-*opt*s cannot be friends.	$\neg opt(X) \wedge \neg opt(Y) \rightarrow \neg hasF(X, Y)$
A pair of non-*opt* friends is impossible.	$\neg opt(X) \wedge \neg opt(Y) \wedge hasF(X, Y) \rightarrow \square$
A pair is either non-friendly or at least one of them is *opt*.	$\blacksquare \rightarrow \neg hasF(X, Y) \vee opt(X) \vee opt(Y)$

- Recall: an empty conjunction (denoted by $\blacksquare$) is true, and an empty disjunction (denoted by $\square$) is false, as $true \wedge A \equiv A$ and $false \vee A \equiv A$.
- In general: you can place any subset of literals into the RHS disjunction and the remaining literals, each negated, into the LHS conjunction.

From propositional resolution to FOL resolution

Assume we have the following clauses:

```
-opt(s).                    % s is non-optimistic.                    (1)
-opt(m).                    % m is non-optimistic.                    (2)
-hasF(s,m)+opt(m)+opt(s).   % if s has m as a friend, either m or s is opt (3')
```

- Given (1)–(3'), can you deduce something using resolution?
- Yes, one can deduce `-hasF(s,m)` using (propositional) resolution.
- What if we consider this FOL clause instead of (3'):
  `-hasF(X,Y)+opt(Y)+opt(X) % if X has Y as a friend, either Y or X is opt` (3)
- Obviously, (3') is a special case of (3), i.e. (3') follows from (3).
- Substitutions for variables `X` and `Y` are obtained through *unification*, a two-way pattern matching algorithm.
- Unification is an essential component of FOL resolution.

FOL resolution – a small example

FOL resolution combines prop. resolution with minimal specialization, e.g.

```
-opt(s).                          (1)
-opt(m).                          (2)
-hasF(X,Y) +opt(Y) +opt(X).       (3)
```

Perform a FOL resolution step between literals (3)#2 and (2)#1:

- find a minimal substitution σ that makes the (unsigned) atomic formulas `opt(Y)` and `opt(m)` the same: $\sigma = \{\texttt{Y} \leftarrow \texttt{m}\}$
- apply σ to the *whole* (3) and (2), resulting in opposite literals:
  (1'): `-hasF(X,m)+opt(m)+opt(X)` and (3'): `-opt(m)`
- perform propositional resolution, producing:

  ```
  -hasF(X,m) +opt(X).             (4)
  ```

  (Is this valid? Yes: if the non-opt m is *x*'s friend, then *x* is an optimist!)
- Next, resolve (4)#2 and (2)#1, $\sigma = \{\texttt{X} \leftarrow \texttt{s}\}$ producing:

  ```
  -hasF(s,m).                     (5)
  ```
- Each time we use a clause, we must rename all its vars systematically
- Similar two-step deductions result in: `-hasF(s,s)`, `-hasF(m,s)`, `-hasF(m,m)`

Unification – making two terms the same

- Propositional resolution requires two clauses with opposite literals `+A` and `-B` where the atomic formula `A` is identical to `B`
- FOL resolution has a weaker requirement: `A` and `B` should be *unifiable*: there should be a substitution σ of variables with terms, such that $\texttt{A}\sigma = \texttt{B}\sigma$ ($\texttt{A}\sigma$ denotes the formula obtained from `A` by applying substitution σ)
- A substitution replaces all occurrences of certain variables with arbitrary terms (possibly other variables)
  - $\sigma = \{\texttt{X} \leftarrow \texttt{b},\ \texttt{Y} \leftarrow \texttt{Z}\},\ \texttt{A} = \texttt{hasF}(\texttt{X},\texttt{Y}),\ \texttt{A}\sigma = \texttt{hasF}(\texttt{b},\texttt{Z})$
  - $\sigma = \{\texttt{X} \leftarrow \texttt{a}\},\ \texttt{A} = \texttt{hasF}(\texttt{m}(\texttt{X}),\texttt{X}),\ \texttt{A}\sigma = \texttt{hasF}(\texttt{m}(\texttt{a}),\texttt{a})$
- Example unification: formulas `A=hasF(a, X)` and `B=hasF(Y, b)` are unifiable using the substitution $\sigma = mgu(\texttt{A},\texttt{B}) = \{\texttt{X} \leftarrow \texttt{b},\ \texttt{Y} \leftarrow \texttt{a}\}$
- If there are multiple substitutions σ for which $\texttt{A}\sigma = \texttt{B}\sigma$, resolution uses the most general unifier, hence the abbreviation *mgu*
- Example: atomic formulas `p(X, X)` and `p(U, V)` are unifiable using the substitution $\sigma = \{\texttt{X} \leftarrow \texttt{U},\ \texttt{V} \leftarrow \texttt{U}\}$ – `U` is not substituted further
  - $\sigma' = \{\texttt{X} \leftarrow \texttt{a},\ \texttt{V} \leftarrow \texttt{a}, \texttt{U} \leftarrow \texttt{a}\}$ is also a unifier, but not a *mgu*
  - The *mgu* is unique, except for variable renaming:
    $\sigma_1 = \{\texttt{X} \leftarrow \texttt{V},\ \texttt{U} \leftarrow \texttt{V}\}$ and $\sigma_2 = \{\texttt{V} \leftarrow \texttt{X},\ \texttt{U} \leftarrow \texttt{X}\}$ are also *mgu*'s

FOL resolution – an example

- In Prop. Logic: $+a \quad \underline{-b}$
  $\underline{+b} \quad -c \quad \Rightarrow +a \quad -c$
- In FOL: $+a(x,0)\underline{-b(x,2)}$
  $\underline{+b(1,y)} - c(y) \Rightarrow +a(1,0) - c(2)$
- Detailed steps:
  1. find substitution $\sigma = mgu(b(x,2), b(1,y)) = \{x \leftarrow 1, y \leftarrow 2\}$
     (note that in general not all variables are necessarily substituted)
  2. apply substitution σ to both clauses
     (vars are universally quantified – substitution is a valid inference):

     $+a(1,0)\underline{-b(1,2)}$
     $\underline{+b(1,2)} - c(2)$
  3. finally, apply propositional resolution, to obtain the resolvent:

     $\Rightarrow +a(1,0) - c(2)$

The resolution inference rule for FOL

- Resolution takes two clauses as input:

  $$C = L_1 \dots L_n \text{ and } D = M_1 \dots M_k$$

  where literals $L_i = \pm A$ and $M_j = \pm B$ have opposite signs,
  and their atomic formulas are unifiable: $\sigma = mgu(A, B)$
- Under the above conditions the resolution inference rule can be applied to C and D and results in the new clause

  $$(L_1 \dots L_{i-1}\, L_{i+1} \dots L_n\, M_1 \dots M_{j-1}\, M_{j+1} \dots M_n)\sigma$$

  obtained by
  - taking the union of the literals of clauses C and D
  - removing the literals L_i and M_j (the ones we resolve upon)
  - applying the substitution σ to the remaining literals
- As specialization (substitution of univ. quantified vars) and propositional resolution are sound operations, FOL resolution is also sound

The factoring inference rule for FOL (ADVANCED)

- For full FOL the resolution rule is not enough to obtain a complete proof system, one needs one more simple rule, called factoring – if there are two literals in a clause that are unifiable, you can replace them by a single literal, their unified form.
- The factoring deduction rule:
  - example in Propositional Logic: $+a+a-b \Rightarrow +a-b$
    here this is "automatic", as clauses are considered sets of literals.
  - example in FOL: $+a(x,2)+a(1,y)-b(x,y) \Rightarrow +a(1,2)-b(1,2)$
  - in general: factoring takes a clause with two unifiable literals and produces a clause with these two literals merged:
    $L_1 \ldots L_n \Rightarrow (L_1 \ldots L_{j-1}\, L_{j+1} \ldots L_n)\sigma$ where $\sigma = mgu(L_i, L_j)$
- For the subset of FOL used in Prolog, this rule is not required, hence it is not discussed further.
- Ancestor resolution (see later) is an alternative to factoring, when implementing a complete FOL theorem prover using Prolog technology.

Resolution: Susan's puzzle

Recall some formulas from slide 57:

7. If x's father or mother is an optimist, so is x, for any x
8. If x has a non-optimist friend, then x is an optimist, for any x.
11. Susan's maternal grandmother has Susan's paternal grandm. as a friend.

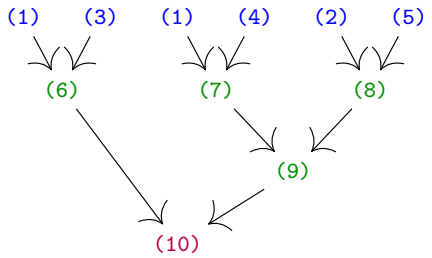
Let us consider a variant of the above example:
We use the `hasP/2` (has parent) pred. instead of father and mother functions.
Also, we replace 11 by 21: Susan's mother has Susan's father as a friend.

Let's formalize the above statements 7, 8 and 21:

7. x is an optimist if x has a parent who is an optimist.
   ```
   +opt(X) -hasP(X, P) -opt(P).   (1)
   ```
8. x is an optimist if x has a friend who is not an optimist.
   ```
   +opt(X) -hasF(X, F) +opt(F).   (2)
   ```
21. Susan's (`s`'s) parents are `m` and `f`, and `m` has `f` as her friend.
   ```
   +hasP(s, m).   (3)
   +hasP(s, f).   (4)
   +hasF(m, f).   (5)
   ```


A resolution proof of the "optimist" example

- Initial clauses:
  ```
  (1) +opt(X) -hasP(X, P) -opt(P).
  (2) +opt(X) -hasF(X, F) +opt(F).
  (3) +hasP(s, m). (4) +hasP(s, f). (5) +hasF(m, f).
  ```
- A possible resolution proof that `(1)`, ..., `(5)` implies `opt(s)`:
  ```
  (1) + (3)    (6) +opt(s) -opt(m).      % s is opt if m is opt
  (1) + (4)    (7) +opt(s) -opt(f).      % s is opt if f is opt
  (2) + (5)    (8) +opt(m) +opt(f).      % m is opt if f is not opt
  (7) + (8)    (9) +opt(s) +opt(m).      % s is opt if m is not opt
  (6) + (9)   (10) +opt(s).              % s is opt
  ```
- The proof as a tree:



Is it feasible to deduce all consequences of a set of clauses?

- Can a clause be resolved with itself?
- Answer: it depends. . . on what kind of logic we use:
  - In propositional logic: **no**, as a clause . . . `+a`. . . `-a`. . . is meaningless
  - In FOL: **yes**, see e.g. clause `(1)` from the previous slide

  `(1)` `+opt(X) -hasP(X, P) -opt(P).`
- Resolve `(1)` with a copy of itself `(1')`:

  `(1')` `+opt(Y) -hasP(Y, Q) -opt(Q).`

  resolving literals `(1)#3` and `(1')#1`, using substitution $\{\texttt{Y} \leftarrow \texttt{P}\}$
- The resolvent:

  `(2)` `+opt(X) -hasP(X, P) -hasP(P, Q) -opt(Q).`

  Read as: "`X` is an optimist if `X` has an optimist grandparent (`Q`)."
- One can keep resolving the output of the previous step with `(1)`, obtaining clauses that describe valid and useful consequences of `(1)`:
  "`X` is an optimist if `X` has an optimist great-grandparent."...
  "`X` is an optimist if `X` has an optimist n^{th} ancestor."
- One can thus infer infinitely many clauses from $\{$`(1)`$\}$

Indirect resolution proofs

- Inferring all consequences of a set of clauses is not a viable task in FOL
- That is why we focus on more focused indirect proofs
- Given a premise U and a consequence V, to prove $(U \to V)$ indirectly:
  - we assume $\neg(U \to V)$, i.e. $U \wedge \neg V$
  - we show that this leads to contradiction, i.e. $U \wedge \neg V \equiv$ false
- What is the truth value of an empty clause (empty disjunction)? false
- The indirect resolution proof of $(U \to V)$ consists of the following steps:
  - convert both U and $\neg V$ to (two) sets of clauses
  - take the union of the two sets and perform resolution (aiming at getting the shortest clauses possible)
  - when an empty clause is reached, the proof is completed
- To prove that clauses `(1)`, ..., `(5)` from page 69 imply `opt(s)`:
  - add $\neg$ `opt(s)` $\equiv$ `-opt(s)` as clause `(10)` (this is the initial goal clause)
  - deduce an empty clause from the set { `(1)`, ..., `(5)`, `(10)`} using resolution

The indirect resolution proof of the "optimist" example

- Initial clauses (so called program clauses: `(1)`–`(5)`, goal clause: `(10)`)

```
(1) +opt(X) -hasP(X, P) -opt(P).
(2) +opt(X) -hasF(X, F) +opt(F).
(3) +hasP(s, m).
(4) +hasP(s, f).
(5) +hasF(m, f).
(10) -opt(s).
```

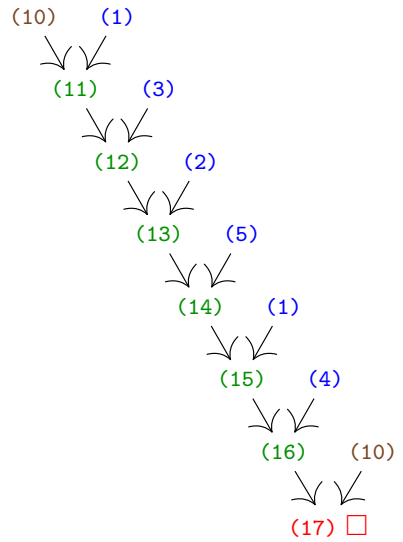
- A possible resolution proof that `(1)`, ..., `(5)`, `(10)` lead to contradiction:

```
                (10) -opt(s).                 % s is non-opt
(10) + (1)      (11) -hasP(s, U) -opt(U).     % all parents of s are non-opt
(11) + (3)      (12) -opt(m).                 % m is non-opt
(12) + (2)      (13) -hasF(m, V) +opt(V).     % all friends of m are opt
(13) + (5)      (14) +opt(f).                 % f is opt
(14) + (1)      (15) +opt(Y) -hasP(Y, f).     % all children of f are opt
(15) + (4)      (16) +opt(s).                 % s is opt
(16) + (10)     (17) □                        % contradiction
```

(Recall that □ denotes an empty clause, i.e. an empty disjunction $\equiv$ false)

The structure of the indirect "optimist" proof

- A linear resolution step is when a goal clause is resolved with a program clause, producing a new goal clause
- All steps in this proof, except for the last, are linear
- The last step is an example of a so called ancestor resolution, as `(16)` is resolved with one of its ancestors in the proof tree, `(10)`
- In Prolog, only linear resolution steps are allowed



Finding optimists on the island

```
(1) +opt(X) -hasP(X, P) -opt(P).        (3) +hasP(s, m).
(2) +opt(X) -hasF(X, F) +opt(F).        (4) +hasP(s, f).
                                        (5) +hasF(m, f).
```

- Let's try to prove indirectly that `(1)`–`(5)` imply $S = \exists z.opt(z)$. Negate S: $\neg S \equiv \neg\exists z.opt(z) \equiv \forall z.\neg opt(z)$, in clausal form: `(10) -opt(Z).`
- A resolution proof showing that `(1)`, ..., `(5)`, `(10)` is contradictory:

```
                (10) -opt(Z).                 % Z is non-opt
(10) + (1)      (11) -hasP(Z, U) -opt(U).     % all parents of Z are non-opt
(11) + (3)           +hasP(s, m).             % { Z = s , U = m}
                (12) -opt(m).                 % m is non-opt
(12) + (2)      (13) -hasF(m, V) +opt(V).     % all friends of m are opt
(13) + (5)      (14) +opt(f).                 % f is opt
-------------------------------------------------------------------
(14) + (10)          -opt(Z').                % { Z' = f }
                (15) □                        % contradiction
```

- We used two instances of the indirect assumption `-opt(Z)`: `Z=s` and `Z'=f`. Thus `(1)`–`(5)` is in contradiction with ($\neg$ `opt(s)` $\wedge\neg$ `opt(f)`).
- Hence `(1)`–`(5)` implies $\neg($ `opt(s)` $\wedge\neg$ `opt(f)`$) \equiv$ `opt(s)` $\vee$ `opt(f)`, i.e. one of `s` and `f` has to be an optimist. Can we get a stronger result?

Finding out who are “optimists”, continued

- Let’s modify the ending of the previous proof (from the red line) so that only a single instantiation of `(10)` is used:

```
(1) +opt(X) -hasP(X, P) -opt(P).          (4) +hasP(s, f).
(2) +opt(X) -hasF(X, F) +opt(F).          (5) +hasF(m, f).
(3) +hasP(s, m).                          (10) -opt(Z).

                (10) -opt(Z).             % Z is non-opt
(10) + (1)      (11) -hasP(Z, U) -opt(U). % all parents of Z are non-opt
(11) + (3)           +hasP(s, m).         % { Z = s , U = m}
                (12) -opt(m).             % m is non-opt
(12) + (2)      (13) -hasF(m, V) +opt(V). % all friends of m are opt
(13) + (5)      (14) +opt(f).             % f is opt
-------------------------------------------------------------------
(14) + (1)      (15) +opt(Y) -hasP(Y, f). % all children of f are opt
(15) + (4)      (16) +opt(s).             % s is opt
(16) + (10)     (17) □                    % { Z’ = s }      contradiction
```

- Here we still used `(10) -opt(Z)` twice, but now in both cases with `Z = s`. Hence we can conclude that `opt(s)` follows from `(1)`–`(5)`.

Finding out who is an “optimist” using the `answer` literal

- We add a special `-answer(Z)` literal to the goal clause
- This literal does not take part in reasoning, it just stores the answer
- Initial clauses:

```
(1) +opt(X) -hasP(X, P) -opt(P).          (4)  +hasP(s, f).
(2) +opt(X) -hasF(X, F) +opt(F).          (5)  +hasF(m, f).
(3) +hasP(s, m).                          (10) -opt(Z)-answer(Z).
```

- The proof:

```
                (10) -opt(Z)              -answer(Z).
(10) + (1)      (11) -hasP(Z, U) -opt(U)  -answer(Z).
(11) + (3)      (12) -opt(m)              -answer(s).
(12) + (2)      (13) -hasF(m, V) +opt(V)  -answer(s).
(13) + (5)      (14) +opt(f)              -answer(s).
(14) + (1)      (15) +opt(Y) -hasP(Y, f)  -answer(s).
(15) + (4)      (16) +opt(s)              -answer(s).
(16) + (10)     (17)                      -answer(s).
```

- The proof ends when only the answer literal is left (cf. empty clause)
- The argument of the answer literal shows the answer: `s`
- Using alternative proofs multiple answers can be obtained

From resolution to Prolog

- The base resolution algorithm leaves several things open:
  - how are the two clauses to be resolved upon selected?
  - how are the literals selected?
- Moving towards Prolog, we now view
  - a Conjunctive NF as a sequence (rather than a set) of clauses
  - a clause as a sequence of literals
- To make reasoning faster, we only allow a subset of FOL clauses: those with at most one positive literal (Definite or Horn clauses)
  - The four kinds of Horn clauses:

Rule: exactly 1 pos lit, $\geq$ 1 neg lits	`(1) +opt(X)-hasP(X, P)-opt(P).`
Fact: exactly 1 pos lit, no neg lits	`(3) +hasP(s, m).`
Goal: no pos lits, $\geq$ 1 neg lits	`(10) -opt(Z).`
Empty: no pos lits, no neg lits	`(17)` □.

    (An empty clause can only occur as the final goal clause)
  - Positive literals are written first (and are called the clause head)
  - In our Susan example the only non-Horn clause was:

```
(2) +opt(X)-hasF(X, F)+opt(F).
```


From resolution to Prolog (ctd.)

```
(o1) +opt(X) -hasP(X, P) -opt(P).         (p1) +hasP(s, f).
(o2) +opt(gm).                            (p2) +hasP(s, m).
                                          (p3) +hasP(m, gm).
```

- Rules and facts define boolean functions, called predicates or procedures. A rule has a head and a body. A fact has a head and no body. A body is a sequence of goals (also called procedure calls).
- Rules and facts are grouped into procedures, based on their functor (`F/N`, where `F` is the name of the clause head, and `N` is the # of args). E.g. procedure `opt/1` contains `(o1)`–`(o2)`, proc. `hasP/2` contains `(p1)`–`(p3)`.
- A goal clause is the same as a body: a list of negative literals or goals. The literal `-opt(s)` is a call of the `opt` procedure, shown above.
- In this example `s` acts as the *actual*, and `X` as the *formal* parameter of the procedure, *unification* is the means for parameter passing
- A resolution step can be viewed as a macro expansion: replace `-opt(s)` by the body of rule `(o1)` with subst. $\{$`X` $\leftarrow$ `s`$\}$: `-hasP(s, P) -opt(P)`
- If multiple clause heads match a call, a so called choice point is created, choices are explored top-to-bottom via backtracking

Example: proving that Susan is an optimist, initial steps

```
(o1) +opt(X) -hasP(X, P) -opt(P).      (p1) +hasP(s, f).
(o2) +opt(gm).                         (p2) +hasP(s, m).
                                       (p3) +hasP(m, gm).
```

Proving that Susan (s) is an optimist, goal: (g1) -opt(s)

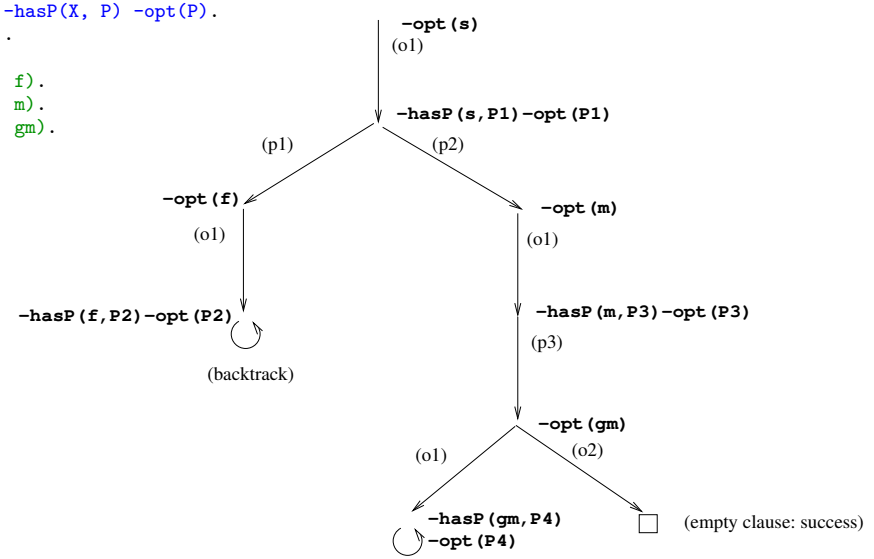
- **Step 1**, matching clause heads: (o1)
- resolve (g1) with copy 1 of (o1), subst. $\{X_1 \leftarrow s\}$ new goal clause: (g2) -hasP(s, P_1) -opt(P_1).
- **Step 2**, matching clauses: (p1), (p2); create CHoice Point 1, storing the goal (g2) and the list of choices: [p1,p2]
- resolve (g2) with (p1), subst. $\{P_1 \leftarrow f\}$ new goal clause: (g3) -opt(f).
- **Step 3**, single matching clause head: (o1), no CHP created
- resolve (g3) with copy 2 of (o1), subst. $\{X_2 \leftarrow f\}$ new goal clause: (g4) -hasP(f, P_2) -opt(P_2).
- **Step 4**, no matching clauses, backtrack to CHP 1, remove branch p1, leaving [p2][1]. Go back to (g2), resolve it now with (p2), subst. $\{P_1 \leftarrow m\}$, new goal clause: (g5) -opt(m).

[1] As this is the last choice, CHP 1 is removed here.

Graphical representation of the resolution search tree

```
(o1) +opt(X) -hasP(X, P) -opt(P).
(o2) +opt(gm).

(p1) +hasP(s, f).
(p2) +hasP(s, m).
(p3) +hasP(m, gm).
```



Prolog as a resolution theorem prover

- Recall the two kinds of clauses: the premises (program clauses) and the goal clause (the negation of the conclusion to be proved)
- Prolog execution uses the following indirect resolution algorithm:
  1. If the goal clause is empty, exit with success (of the indirect proof)
  2. Otherwise, find all program clauses whose first literal can be resolved with the first goal literal, scanning top to bottom
  3. If there are > 1 such clauses, create a choice point storing this list of applicable clauses and the current goal clause
  4. If there are ≥ 1 such clauses, resolve the goal clause with the first applicable program clause, make the resolvent the new goal clause, and go to step 1
  5. If there are no such clauses, backtrack:
     - if no choice points are left, exit with failure (of the indirect proof)
     - consider the latest choice point (choice points form a stack), restore the goal clause from the choice point, resolve it with the next applicable clause and continue at step 1.

Prolog as a resolution theorem prover

- The Prolog programming language is based on indirect, goal oriented resolution; with the following constraints (recap):
  - the SELECTION of literals is restricted: only the first literals in both clauses can be used for resolution
  - resolution is applied in a *LINEAR* manner: start with the goal, resolve it with a rule or fact, and repeat this for the resolvent
  - only DEFINITE (Horn) Clauses are allowed
- Prolog is thus based on SLD resolution – Selective Linear resolution on Definite clauses

Performing queries using resolution – practice

- Consider the program

```
+hP(a, b).                          (1)
+hP(b, c).                          (2)
+hP(b, d).                          (3)
+hP(d, e).                          (4)

+hGP(Ch, GP) -hP(Ch, P) -hP(P, GP). (5)
```

- Execute the following goals using SLD resolution:

```
-hGP(a,  GP).                       (11)
-hGP(b,  GP).                       (12)
-hGP(d,  GP).                       (13)
-hGP(Ch, e).                        (14)
-hGP(Ch, b).                        (15)
-hGP(Ch, GP).                       (16)
```


Limitations of Prolog

- Equality can not be used in positive literals (clause heads), e.g. these formulas cannot be converted to Prolog:

$$\forall x.(x = s() \leftarrow opt(x)) \qquad \text{(only Susan can be optimistic)}$$

$$\forall x, y.(x + y = y + x) \qquad \text{(addition is commutative)}$$

- Consequence: function symbols become data constructors, e.g.

```
| ?- X = 1+2*3.            X = 1+2*3 ?
| ?- X is 1+2*3.           X = 7 ?      % is is a built-in for arithmetic
| ?- X = 1+2*3, Y+Z = X.   X = 1+2*3, Y = 1, Z = 2*3 ?
```

- Prolog unification does not do the occurs check:
  - FOL resolution prescribes a variable x cannot be unified with a term α, if x occurs in α.
  - This costly check is practically useless in Prolog and by default is not performed by Prolog systems. (However, there is a built-in predicate `unify_with_occurs_check`, to perform this.)