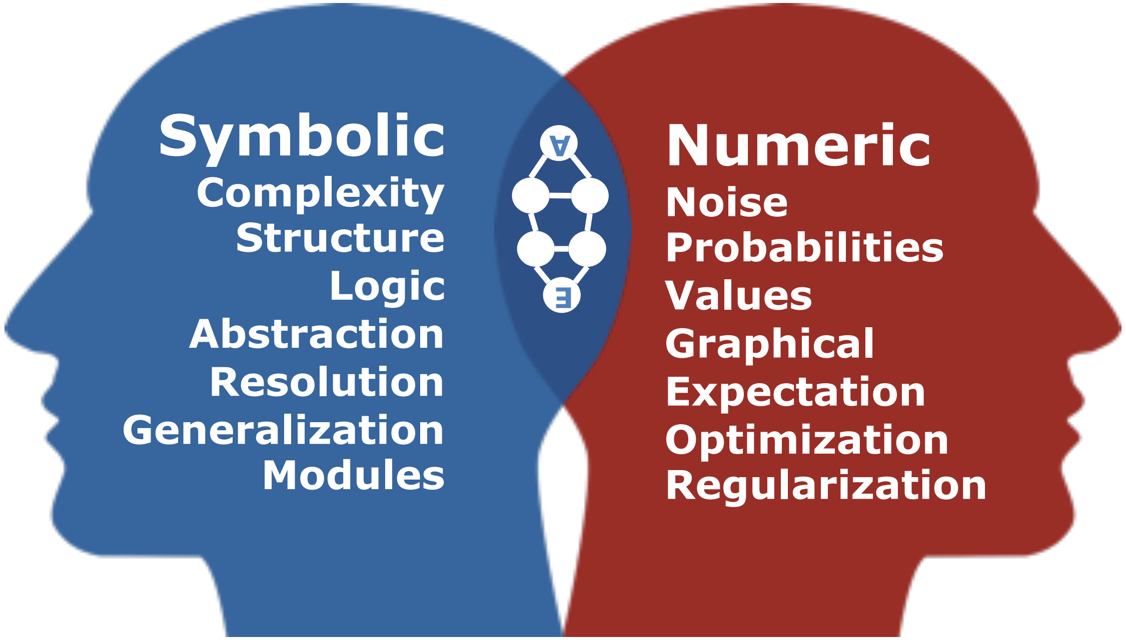


Statistical Relational AI

(First-order) Logic (Programming)

Kristian
Kersting

Thanks to Vincent Conitzer, Rina Dechter, Luc De Raedt, Pedro Domingos, Peter Flach, Dieter Fensel, Florian Ficher, Vibhav Gogate, Carlos Guestrin, Daphen Koller, Nir Friedman, Ray Mooney, Sriraam Natarajan, David Poole, Fabrizio Riguzzi, Dan Suciu, Guy van den Broeck, and many others for making their slides publically available

Logic and AI

TECHNISCHE UNIVERSITÄT DARMSTADT

- Would like our AI to have knowledge about the world, and logically draw conclusions from it
- Search algorithms generate successors and evaluate them, but do not “understand” much about the setting
- Example question: is it possible for a chess player to have 8 pawns and 2 queens?
  - Search algorithm could search through tons of states to see if this ever happens, but…

A “roommate” story

- You roommate comes home; he/she is completely wet
- You know the following things:
  - Your roommate is wet
  - If your roommate is wet, it is because of rain, sprinklers, or both
  - If your roommate is wet because of sprinklers, the sprinklers must be on
  - If your roommate is wet because of rain, your roommate must not be carrying the umbrella
  - The umbrella is not in the umbrella holder
  - If the umbrella is not in the umbrella holder, either you must be carrying the umbrella, or your roommate must be carrying the umbrella
  - You are not carrying the umbrella
- **Can you conclude that the sprinklers are on?**
- **Can AI conclude that the sprinklers are on?**

Knowledge base for the story

- RoommateWet
- RoommateWet => (RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers)
- RoommateWetBecauseOfSprinklers => SprinklersOn
- RoommateWetBecauseOfRain => NOT(RoommateCarryingUmbrella)
- UmbrellaGone
- UmbrellaGone => (YouCarryingUmbrella OR RoommateCarryingUmbrella)
- NOT(YouCarryingUmbrella)

Syntax

- What do well-formed sentences in the knowledge base look like?
- A BNF grammar:
- *Symbol* → P, Q, R, …, RoommateWet, …
- *Sentence* → True | False | *Symbol* | NOT(*Sentence*) | (*Sentence* AND *Sentence*) | (*Sentence* OR *Sentence*) | (*Sentence* => *Sentence*)
- We will drop parentheses sometimes, but formally they really should always be there

Semantics

TECHNISCHE UNIVERSITÄT DARMSTADT

- A model specifies which of the proposition symbols are true and which are false
- Given a model, I should be able to tell you whether a sentence is true or false
- Truth table defines semantics of operators:

a	b	NOT(a)	a AND b	a OR b	a => b
false	false	true	false	false	true
false	true	true	false	true	true
true	false	false	false	true	false
true	true	false	true	true	true

- Given a model, can compute truth of sentence recursively with these

Caveats

- TwoIsAnEvenNumber OR ThreeIsAnOddNumber

  is true (not exclusive OR)
- TwoIsAnOddNumber => ThreeIsAnEvenNumber

  is true (if the left side is false it's always true)

All of this is assuming those symbols are assigned their natural values…

Tautologies

TECHNISCHE UNIVERSITÄT DARMSTADT

- A sentence is a tautology if it is true for any setting of its propositional symbols

P	Q	P OR Q	NOT(P) AND NOT(Q)	(P OR Q) OR (NOT(P) AND NOT(Q))
false	false	false	true	true
false	true	true	false	true
true	false	true	false	true
true	true	true	false	true

- (P OR Q) OR (NOT(P) AND NOT(Q)) is a tautology

Is this a tautology?

- (P => Q) OR (Q => P)

Logical equivalences

- Two sentences are logically equivalent if they have the same truth value for every setting of their propositional variables

P	Q	P OR Q	NOT(NOT(P) AND NOT(Q))
false	false	false	false
false	true	true	true
true	false	true	true
true	true	true	true

- P OR Q and NOT(NOT(P) AND NOT(Q)) are logically equivalent
- Tautology = logically equivalent to True

Famous logical equivalences

they can be used for rewriting and simplifying rules

- (a OR b) ≡ (b OR a) *commutatitvity*
- (a AND b) ≡ (b AND a) *commutatitvity*
- ((a AND b) AND c) ≡ (a AND (b AND c)) *associativity*
- ((a OR b) OR c) ≡ (a OR (b OR c)) *associativity*
- NOT(NOT(a)) ≡ a *double-negation elimination*
- (a => b) ≡ (NOT(b) => NOT(a)) *contraposition*
- (a => b) ≡ (NOT(a) OR b) *implication elimination*
- NOT(a AND b) ≡ (NOT(a) OR NOT(b)) *De Morgan*
- NOT(a OR b) ≡ (NOT(a) AND NOT(b)) *De Morgan*
- (a AND (b OR c)) ≡ ((a AND b) OR (a AND c)) *distributitivity*
- (a OR (b AND c)) ≡ ((a OR b) AND (a OR c)) *distributitivity*

Inference

TECHNISCHE UNIVERSITÄT DARMSTADT

- We have a knowledge base of things that we know are true
  - RoommateWetBecauseOfSprinklers
  - RoommateWetBecauseOfSprinklers => SprinklersOn
- Can we conclude that SprinklersOn?
- We say SprinklersOn is entailed by the knowledge base if, for every setting (models) of the propositional variables for which the knowledge base is true, SprinklersOn is also true

RWBOS	SprinklersOn	Knowledge base
false	false	false
false	true	false
true	false	false
true	true	true

- SprinklersOn is entailed!

Simple algorithm for inference

- Want to find out if sentence a is entailed by knowledge base…
- *Go through the possible settings of the propositional variables,*
  - *If knowledge base is true and a is false, return false*
- *Return true*
- Not very efficient: $2^{\#\text{propositional variables}}$ settings

Inconsistent knowledge bases

- Suppose we were careless in how we specified our knowledge base:
  - PetOfRoommateIsABird => PetOfRoommateCanFly
  - PetOfRoommateIsAPenguin => PetOfRoommateIsABird
  - PetOfRoommateIsAPenguin => NOT(PetOfRoommateCanFly)
  - PetOfRoommateIsAPenguin
- It entails both PetOfRoommateCanFly and NOT(PetOfRoommateCanFly)
- Therefore, technically, this knowledge base implies anything: The Moon Is Made Of Cheese

The Moon is Made of Cheese

- PetOfRoommateCanFly AND NOT(PetOfRoommateCanFly)
- PetOfRoommateCanFly, NOT(PetOfRoommateCanFly)
- Now, “a true statement OR anything else” is always true, hence
  - PetOfRoommateCanFly OR MoonMadeOfCheese
- Therefore, MoonMadeOfCheese has to be true.

- Please note that you really put anything there!

So, we justify the Aristotelian claim that “there cannot be contradictions” (The Principle of Non-Contradiction)

Reasoning patterns

- Obtain new sentences directly from some other sentences in the knowledge base according to reasoning patterns
- If we have sentences a and a => b , we can correctly conclude the new sentence b
  - This is called modus ponens
- If we have a AND b , we can correctly conclude a
- All of the logical equivalences from before also give reasoning patterns

Formal proof that the sprinklers are on

1) RoommateWet
2) RoommateWet => (RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers)
3) RoommateWetBecauseOfSprinklers => SprinklersOn
4) RoommateWetBecauseOfRain => NOT(RoommateCarryingUmbrella)
5) UmbrellaGone
6) UmbrellaGone => (YouCarryingUmbrella OR RoommateCarryingUmbrella)
7) NOT(YouCarryingUmbrella)

Knowledge Base

8) YouCarryingUmbrella OR RoommateCarryingUmbrella *(modus ponens on 5 and 6)*
9) NOT(YouCarryingUmbrella) => RoommateCarryingUmbrella *(equivalent to 8)*
10) RoommateCarryingUmbrella *(modus ponens on 7 and 9)*
11) NOT(NOT(RoommateCarryingUmbrella) *(equivalent to 10)*
12) NOT(NOT(RoommateCarryingUmbrella)) => NOT(RoommateWetBecauseOfRain) *(equivalent to 4 by contraposition)*
13) NOT(RoommateWetBecauseOfRain) *(modus ponens on 11 and 12)*
14) RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers *(modus ponens on 1 and 2)*
15) NOT(RoommateWetBecauseOfRain) => RoommateWetBecauseOfSprinklers *(equivalent to 14)*
16) RoommateWetBecauseOfSprinklers *(modus ponens on 13 and 15)*
17) SprinklersOn *(modus ponens on 16 and 3)*

Reasoning about penguins

1) PetOfRoommateIsABird => PetOfRoommateCanFly
2) PetOfRoommateIsAPenguin => PetOfRoommateIsABird
3) PetOfRoommateIsAPenguin => NOT(PetOfRoommateCanFly)
4) PetOfRoommateIsAPenguin
5) PetOfRoommateIsABird *(modus ponens on 4 and 2)*
6) PetOfRoommateCanFly *(modus ponens on 5 and 1)*
7) NOT(PetOfRoommateCanFly) *(modus ponens on 4 and 3)*
8) NOT(PetOfRoommateCanFly) => FALSE *(equivalent to 6)*
9) FALSE *(modus ponens on 7 and 8)*
10) FALSE => TheMoonIsMadeOfCheese *(tautology)*
11) TheMoonIsMadeOfCheese *(modus ponens on 9 and 10)*

Getting more systematic

- Any knowledge base can be written as a single formula in conjunctive normal form (CNF)
  - CNF formula: (… OR … OR …) AND (… OR …) AND …
  - … can be a symbol x, or NOT(x) (these are called literals)
  - Multiple facts in knowledge base are effectively ANDed together

RoommateWet => (RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers)

becomes

(NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers)

Converting story problem to conjunctive normal form

- RoommateWet
  - RoommateWet
- RoommateWet => (RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers)
  - NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers
- RoommateWetBecauseOfSprinklers => SprinklersOn
  - NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn
- RoommateWetBecauseOfRain => NOT(RoommateCarryingUmbrella)
  - NOT(RoommateWetBecauseOfRain) OR NOT(RoommateCarryingUmbrella)
- UmbrellaGone
  - UmbrellaGone
- UmbrellaGone => (YouCarryingUmbrella OR RoommateCarryingUmbrella)
  - NOT(UmbrellaGone) OR YouCarryingUmbrella OR RoommateCarryingUmbrella
- NOT(YouCarryingUmbrella)
  - NOT(YouCarryingUmbrella)

Unit resolution

If we have

- l_1 OR l_2 OR … OR l_k

and

- NOT(l_i)

we can conclude

- l_1 OR l_2 OR … l_{i-1} OR l_{i+1} OR … OR l_k
- Basically modus ponens

Applying resolution to story problem

1) RoommateWet
2) NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers
3) NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn
4) NOT(RoommateWetBecauseOfRain) OR NOT(RoommateCarryingUmbrella)
5) UmbrellaGone
6) NOT(UmbrellaGone) OR YouCarryingUmbrella OR RoommateCarryingUmbrella
7) NOT(YouCarryingUmbrella)
8) NOT(UmbrellaGone) OR RoommateCarryingUmbrella (6,7)
9) RoommateCarryingUmbrella (5,8)
10) NOT(RoommateWetBecauseOfRain) (4,9)
11) NOT(RoommateWet) OR RoommateWetBecauseOfSprinklers (2,10)
12) RoommateWetBecauseOfSprinklers (1,11)
13) SprinklersOn (3,12)

Limitations of unit resolution

- P OR Q
- NOT(P) OR Q
- Can we conclude Q?

(General) resolution

if we have

- l_1 OR l_2 OR … OR l_k
  and
- m_1 OR m_2 OR … OR m_n
  where for some i,j, l_i = NOT(m_j)

we can conclude

- l_1 OR l_2 OR … l_{i-1} OR l_{i+1} OR … OR l_k OR m_1 OR m_2 OR … OR m_{j-1} OR m_{j+1} OR … OR m_n
- Same literal may appear multiple times; remove those

Applying resolution to story problem (more clumsily)

1) RoommateWet
2) NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers
3) NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn
4) NOT(RoommateWetBecauseOfRain) OR NOT(RoommateCarryingUmbrella)
5) UmbrellaGone
6) NOT(UmbrellaGone) OR YouCarryingUmbrella OR RoommateCarryingUmbrella
7) NOT(YouCarryingUmbrella)
8) NOT(RoommateWet) OR RoommateWetBecauseOfRain OR SprinklersOn (2,3)
9) NOT(RoommateCarryingUmbrella) OR NOT(RoommateWet) OR SprinklersOn (4,8)
10) NOT(UmbrellaGone) OR YouCarryingUmbrella OR NOT(RoommateWet) OR SprinklersOn (6,9)
11) YouCarryingUmbrella OR NOT(RoommateWet) OR SprinklersOn (5,10)
12) NOT(RoommateWet) OR SprinklersOn (7,11)
13) SprinklersOn (1,12)

Systematic inference?

- General strategy: if we want to see if sentence a is entailed, add NOT(a) to the knowledge base and see if it becomes inconsistent (we can derive a contradiction)
- CNF formula for modified knowledge base is satisfiable if and only if sentence a is not entailed
  - Satisfiable = there exists a model that makes the modified knowledge base true = modified knowledge base is consistent

Resolution algorithm

- Given formula in conjunctive normal form, **repeat:**
  - **Find** two clauses with complementary literals,
  - **Apply resolution**,
  - **Add resulting clause** (if not already there)
  - **If the empty clause results, formula is not satisfiable**
    - Must have been obtained from P and NOT(P)
  - **Otherwise, if we get stuck** (and we will **eventually**), **the formula is guaranteed to be satisfiable** (proof in a couple of slides and this also illustrates that propositional logic is decidable)

Example

Our knowledge base:

- 1) RoommateWetBecauseOfSprinklers
- 2) NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn

Can we infer SprinklersOn?

- We add:
  - 3) NOT(SprinklersOn)
- From 2) and 3), get
  - 4) NOT(RoommateWetBecauseOfSprinklers)
- From 4) and 1), get empty clause

If we get stuck, why is the formula satisfiable?

- Consider the final set of clauses C
- Construct satisfying assignment as follows:
- Assign truth values to variables in order $x_1, x_2, \ldots, x_n$
- If x_j is the last chance to satisfy a clause (i.e., all the other variables in the clause came earlier and were set the wrong way), then set x_j to satisfy it
  - Otherwise, doesn't matter how it's set
- Suppose this fails (for the first time) at some point, i.e., x_j must be set to true for one last-chance clause and false for another
- These two clauses would have resolved to something involving only up to x_{j-1} (not to the empty clause, of course), which must be satisfied
- But then one of the two clauses must also be satisfied - contradiction

Special case: Horn clauses

- Horn clauses are implications with only positive literals

  x_1 AND x_2 AND x_4 => x_3 AND x_6

  TRUE => x_1

- Try to figure out whether some x_j is entailed
- Simply follow the implications (modus ponens) as far as you can, see if you can reach x_j
- x_j is entailed if and only if it can be reached (can set everything that is not reached to false)
- Can implement this more efficiently by maintaining, for each implication, a count of how many of the left-hand side variables have been reached (we stopped hear)

Limitations of propositional logic

- Some English statements are hard to model in propositional logic:

  "If your roommate is wet because of rain, your roommate must not be carrying **any** umbrella"

- Pathetic attempt at modeling this:

  RoommateWetBecauseOfRain =>
  (NOT(RoommateCarryingUmbrella0) AND
  NOT(RoommateCarryingUmbrella1) AND
  NOT(RoommateCarryingUmbrella2) AND …)

Limitations of propositional logic

- No notion of objects
- No notion of relations among objects
- RoommateCarryingUmbrella0 is instructive **to us**, suggesting
  - there is an object we call Roommate,
  - there is an object we call Umbrella0,
  - there is a relationship Carrying between these two objects
- Formally, none of this meaning is there
  - Might as well have replaced RoommateCarryingUmbrella0 by P

Elements of first-order logic

- Objects: can give these names such as Umbrella0, Person0, John, Earth, …
- Relations: Carrying(., .), IsAnUmbrella(.)
  - Carrying(Person0, Umbrella0), IsUmbrella(Umbrella0)
  - Relations with one object = unary relations = properties
- Functions: Roommate(.)
  - Roommate(Person0)
- Equality: Roommate(Person0) = Person1

Things to note about functions

- It could be that we have a separate name for Roommate(Person0)
- E.g., Roommate(Person0) = Person1
- … but we do not **need** to have such a name

- A function can be applied to any object
- E.g., Roommate(Umbrella0)

Reasoning about many objects at once

- **Variables**: x, y, z, … can refer to multiple objects
- New operators “for all” and “there exists”
  - **Universal quantifier** and **existential quantifier**
- for all x: CompletelyWhite(x) => NOT(PartiallyBlack(x))
  - Completely white objects are never partially black
- there exists x: PartiallyWhite(x) AND PartiallyBlack(x)
  - There exists some object in the world that is partially white and partially black

Practice converting English to first-order logic

- "John has Jane's umbrella"
- Has(John, Umbrella(Jane))
- "John has an umbrella"
- there exists y: (Has(John, y) AND IsUmbrella(y))
- "Anything that has an umbrella is not wet"
- for all x: ((there exists y: (Has(x, y) AND IsUmbrella(y))) => NOT(IsWet(x)))
- "Any person who has an umbrella is not wet"
- for all x: (IsPerson(x) => ((there exists y: (Has(x, y) AND IsUmbrella(y))) => NOT(IsWet(x))))

More practice converting English to first-order logic

- "John has at least two umbrellas"
- there exists x: (there exists y: (Has(John, x) AND IsUmbrella(x) AND Has(John, y) AND IsUmbrella(y) AND NOT(x=y))
- "John has at most two umbrellas"
- for all x, y, z: ((Has(John, x) AND IsUmbrella(x) AND Has(John, y) AND IsUmbrella(y) AND Has(John, z) AND IsUmbrella(z)) => (x=y OR x=z OR y=z))

Even more practice converting English to first-order logic…

- "TUDa's basketball team defeats any other basketball team"
- for all x: ((IsBasketballTeam(x) AND NOT(x=BasketballTeamOf(TUDa))) => Defeats(BasketballTeamOf(TUDa), x))
- "Every team defeats some other team"
- for all x: (IsTeam(x) => (there exists y: (IsTeam(y) AND NOT(x=y) AND Defeats(x,y))))

More realistically…

- “Any basketball team that defeats TUDa’s basketball team in one year will be defeated by TUDa’s basketball team in a future year”
- for all x,y: (IsBasketballTeam(x) AND IsYear(y) AND DefeatsIn(x, BasketballTeamOf(TUDa), y)) => there exists z: (IsYear(z) AND IsLaterThan(z,y) AND DefeatsIn(BasketballTeamOf(TUDa), x, z))

Relationship between universal and existential

- for all x: a
- is equivalent to
- NOT(there exists x: NOT(a))

Something we cannot do in first-order logic

- We are **not** allowed to reason in general about relations and functions
- The following would correspond to higher-order logic (which is more powerful):
- “If John is Jack’s roommate, then any property of John is also a property of Jack’s roommate”
- (John=Roommate(Jack)) => for all p: (p(John) => p(Roommate(Jack)))
- “If a property is inherited by children, then for any thing, if that property is true of it, it must also be true for any child of it”
- for all p: (IsInheritedByChildren(p) => (for all x, y: ((IsChildOf(x,y) AND p(y)) => p(x))))

Axioms and theorems

- Axioms: basic facts about the domain, our “initial” knowledge base
- Theorems: statements that are logically derived from axioms

SUBST

- SUBST replaces one or more variables with something else
- For example:
  - SUBST({x/John}, IsHealthy(x) => NOT(HasACold(x))) gives us
  - IsHealthy(John) => NOT(HasACold(John))

Instantiating quantifiers

- From
- for all x: a
- we can obtain
- SUBST({x/g}, a)

- From
- there exists x: a
- we can obtain
- SUBST({x/k}, a)
- where k is a constant that does not appear elsewhere in the knowledge base (Skolem constant)
- Don’t need original sentence anymore

Instantiating existentials after universals

- for all x: there exists y: IsParentOf(y,x)
- WRONG: for all x: IsParentOf(k, x)
- RIGHT: for all x: IsParentOf(k(x), x)
- Introduces a new function (Skolem function)
- … again, assuming k has not been used previously

Generalized modus ponens

- for all x: Loves(John, x)
  - John loves every thing
- for all y: (Loves(y, Jane) => FeelsAppreciatedBy(Jane, y))
  - Jane feels appreciated by every thing that loves her
- Can infer from this:
- FeelsAppreciatedBy(Jane, John)

- Here, we used the substitution {x/Jane, y/John}
  - Note we used different variables for the different sentences
- General UNIFY algorithms for finding a good substitution

Keeping things as general as possible in unification

- Consider EdibleByWith
  - e.g., EdibleByWith(Soup, John, Spoon) – John can eat soup with a spoon
- for all x: for all y: EdibleByWith(Bread, x, y)
  - Anything can eat bread with anything
- for all u: for all v: (EdibleByWith(u, v, Spoon) => CanBeServedInBowlTo(u,v))
  - Anything that is edible with a spoon by something can be served in a bowl to that something
- Substitution: {x/z, y/Spoon, u/Bread, v/z}
- Gives: for all z: CanBeServedInBowlTo(Bread, z)
- Alternative substitution {x/John, y/Spoon, u/Bread, v/John} would only have given CanBeServedInBowlTo(Bread, John), which is not as general

Resolution for first-order logic

- for all x: (NOT(Knows(John, x)) OR IsMean(x) OR Loves(John, x))
  - John loves everything he knows, with the possible exception of mean things
- for all y: (Loves(Jane, y) OR Knows(y, Jane))
  - Jane loves everything that does not know her
- What can we unify? What can we conclude?
- Use the substitution: {x/Jane, y/John}
- Get: IsMean(Jane) OR Loves(John, Jane) OR Loves(Jane, John)
- Complete (i.e., if not satisfiable, will find a proof of this), **if** we can remove literals that are duplicates after unification
  - Also need to put everything in canonical form first

Notes on inference in first-order logic

- Deciding whether a sentence is entailed is semidecidable: there are algorithms that will eventually produce a proof of any entailed sentence
- It is **NOT** decidable: we cannot always conclude that a sentence is not entailed

(Extremely informal statement of) Gödel's Incompleteness Theorem

- First-order logic is not rich enough to model basic arithmetic
- For any consistent system of axioms that is rich enough to capture basic arithmetic (in particular, mathematical induction), there exist true sentences that cannot be proved from those axioms

(Extremely informal statement of) Gödel's Incompleteness Theorem

Quite informally stated consequence:

"Not everything in the world is provable"

A more challenging exercise

- Suppose:
  - There are exactly 3 objects in the world,
  - If x is the spouse of y, then y is the spouse of x (spouse is a function, i.e., everything has a spouse)
- Prove:
  - Something is its own spouse

More challenging exercise

- there exist x, y, z: (NOT(x=y) AND NOT(x=z) AND NOT (y=z))
- for all w, x, y, z: (w=x OR w=y OR w=z OR x=y OR x=z OR y=z)
- for all x, y: ((Spouse(x)=y) => (Spouse(y)=x))
- for all x, y: ((Spouse(x)=y) => NOT(x=y)) *(for the sake of contradiction)*
- *Try to do this on the board…*

Umbrellas in first-order logic

- You know the following things:
  - You have exactly one other person living in your house, who is wet
  - If a person is wet, it is because of the rain, the sprinklers, or both
  - If a person is wet because of the sprinklers, the sprinklers must be on
  - If a person is wet because of rain, that person must not be carrying any umbrella
  - There is an umbrella that “lives in” your house, which is not in its house
  - An umbrella that is not in its house must be carried by some person who lives in that house
  - You are not carrying any umbrella
- Can you conclude that the sprinklers are on?

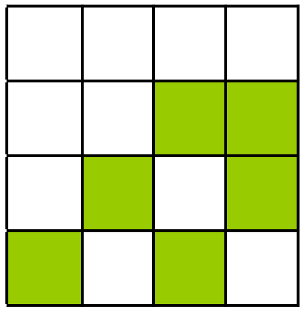
Applications

TECHNISCHE UNIVERSITÄT DARMSTADT

- Some serious novel mathematical results proved
- Verification of hardware and software
  - Prove outputs satisfy required properties for all inputs
- Synthesis of hardware and software
  - Try to prove that there exists a program satisfying such and such properties, in a constructive way
- And currently, a lot of interest in combining deep learning and logical reasoning / theorem provers

Logic (Programming): How to program using logic

(Person, Food)



Person	Food
sam	dal
sam	curry
josie	samosas
josie	curry
rajiv	burgers
rajiv	dal

- The above shows an ordinary constraint between two variables: Person and Food
- Prolog makes you name this constraint.
  Here’s a program that defines it:
  - eats(sam, dal). eats(josie, samosas).
  - eats(sam, curry). eats(josie, curry).
  - eats(rajiv, burgers). eats(rajiv, dal). …
- Now it acts like a subroutine! At the Prolog prompt you can type
  - eats(Person1, Food1). % constraint over two variables
  - eats(Person2, Food2). % constraint over two **other** variables

Simple constraints in Prolog

- Here's a program defining the "eats" constraint:
  - eats(sam, dal). eats(josie, samosas).
  - eats(sam, curry). eats(josie, curry).
  - eats(rajiv, burgers). eats(rajiv, dal). …
  - Now at the Prolog prompt you can type
    - eats(Person1, Food1). % constraint over two variables
    - eats(Person2, Food2). % constraint over two **other** variables

- To say that Person1 and Person2 must eat a common food, conjoin two constraints with a **comma**:
  - eats(Person1, Food), eats(Person2, Food).
  - Prolog gives you possible solutions:
    - Person1=sam, Person2=josie, Food=curry
    - Person1=josie, Person2=sam, Food=curry …

Actually, it will start with solutions where Person1=sam, Person2=sam. How to fix?

Queries in Prolog

The things you type at the prompt are called "queries."

- Prolog answers a query as "Yes" or "No"
  according to whether it can find a satisfying assignment.
- If it finds an assignment, it prints the first one before printing "Yes."
- You can press Enter to accept it, in which case you're done,
  or ";" to reject it, causing Prolog to backtrack and look for another.

  - eats(Person1, Food1). % constraint over two variables
  - eats(Person2, Food2). % constraint over two **other** variables
- eats(Person1, Food), eats(Person2, Food).
- Prolog gives you possible solutions:
  - Person1=sam, Person2=josie, Food=curry [press ";"]
  - Person1=josie, Person2=sam, Food=curry …

Constants vs. Variables

TECHNISCHE UNIVERSITÄT DARMSTADT

- Here's a program defining the "eats" constraint:
  - eats(sam, dal). eats(josie, samosas).
  - eats(sam, curry). eats(josie, curry).
  - eats(rajiv, burgers). …
  - Now at the Prolog prompt you can type
    - eats(Person1, Food1). % constraint over two variables
    - eats(Person2, Food2). % constraint over two **other** variables

- Nothing stops you from putting constants into constraints:
  - eats(josie, Food). % what Food does Josie eat? (2 answers)
  - eats(Person, curry). % what Person eats curry? (2 answers)
  - eats(josie, Food), eats(Person, Food). % who'll share what with Josie?
    - Food=curry, Person=sam

Constants vs. Variables

- Variables start with A,B,…Z or underscore:
  - Food, Person, Person2, _G123
- Constant “atoms” start with a,b,…z or appear in single quotes:
  - josie, curry, ’CS325’
  - Other kinds of constants besides atoms:
    - Integers -7, real numbers 3.14159, the empty list []
    - eats(josie,curry) is technically a constant structure

- Nothing stops you from putting constants into constraints:
  - eats(josie, Food). % what Food does Josie eat? (2 answers)
  - eats(Person, curry). % what Person eats curry? (2 answers)
  - eats(josie, Food), eats(Person, Food). % who’ll share what with Josie?
    - Food=curry, Person=sam

Rules in Prolog

TECHNISCHE UNIVERSITÄT DARMSTADT

- Let's augment our program with a new constraint:

```
eats(sam, dal).                 eats(josie, samosas).
eats(sam, curry).               eats(josie, curry).
eats(rajiv, burgers).           eats(rajiv, dal).
compatible(Person1, Person2) :- eats(Person1, Food),
                                eats(Person2, Food).
```

head

body

means "if" – it's supposed to look like "←"

  - "Person1 and Person2 are compatible if there exists some Food that they both eat."
  - "One way to satisfy the head of this rule is to satisfy the body."
  - You type the query: compatible(rajiv, X). Prolog answers: X=sam.
    - Prolog doesn't report that Person1=rajiv, Person2=sam, Food=dal. These act like local variables in the rule. It already forgot about them.

The Prolog solver

- Prolog's solver is incredibly simple.
- eats(sam,X).
  - Iterates in order through the program's "eats" clauses.
  - First one to match is eats(sam,dal).
    so it returns with X=dal.
  - If you hit semicolon, it backtracks and continues:
    Next match is eats(sam,curry).
    so it returns with X=curry.

The Prolog solver

- Prolog's solver is incredibly simple.
- eats(sam,X).

- eats(sam,X), eats(josie,X).
  - It satisfies 1st constraint with X=dal. Now X is assigned.
  - Now to satisfy 2nd constraint, it must prove eats(josie,dal). No!
  - So it backs up to 1st constraint & tries X=curry (sam's other food).
  - Now it has to prove eats(josie,curry). Yes!
  - So it is able to return X=curry. What if you now hit semicolon?

- eats(sam,X), eats(Companion, X).
  - What happens here?
  - What variable ordering is being used? Where did it come from?
  - What value ordering is being used? Where did it come from?

The Prolog solver

- Prolog's solver is incredibly simple.
- eats(sam,X).
- eats(sam,X), eats(josie,X).
- eats(sam,X), eats(Companion, X).
- compatible(sam,Companion).
  - This time, first clause that matches is
    compatible(Person1, Person2) :- eats(Person1, Food),
    eats(Person2, Food).
  - "Head" of clause matches with Person1=sam, Person2=Companion.
  - So now we need to satisfy "body" of clause:
    eats(sam,Food), eats(Companion,Food).
    Look familiar?
  - We get Companion=rajiv.

The Prolog solver

- Prolog's solver is incredibly simple.
- eats(sam,X).
- eats(sam,X), eats(josie,X).
- eats(sam,X), eats(Companion, X).
- compatible(sam,Companion).

- compatible(sam,Companion), female(Companion).
  - **compatible(Person1, Person2) :- eats(Person1, Food),**
    eats(Person2, Food).
  - Our first try at satisfying 1st constraint is Companion=rajiv (as before).
    - But then 2nd constraint is female(rajiv). which is presumably false.
  - So we backtrack and look for a different satisfying assignment of the first constraint: Companion=josie.
    - Now 2nd constraint is female(josie). which is presumably true.
    - We backtracked into this **compatible** clause (food) & retried it.
    - No need yet to move on to the next **compatible** clause (movies).

Prolog as a database language

TECHNISCHE UNIVERSITÄT DARMSTADT

- The various eats(…, …) facts can be regarded as rows in a database (2-column database in this case).
- Standard relational database operations:

```
• eats(X,dal).                                   % select
• edible(Object) :- eats(Someone, Object).       % project
• parent(X,Y) :- mother(X,Y).                    % union
  parent(X,Y) :- father(X,Y).
• sister_in_law(X,Z) :- sister(X,Y), married(Y,Z).   % join
```

- Why the heck does anyone still use SQL? Beats me.
- Warning: Prolog's backtracking strategy can be inefficient.
  - But we can keep the little language illustrated above ("Datalog") and instead compile into optimized query plans, just as for SQL.

Recursive queries

TECHNISCHE UNIVERSITÄT DARMSTADT

- Prolog allows recursive queries (SQL doesn't).
- Who's married to their boss?
  - boss(X,Y), married(X,Y).
- Who's married to their boss's boss?
  - boss(X,Y), boss(Y,Z), married(X,Z).
- Who's married to their boss's boss's boss?
  - Okay, this is getting silly. Let's do the general case.
- Who's married to someone above them?
  - above(X,X).
  - above(X,Y) :- boss(X,Underling), above(Underling,Y).
  - above(X,Y), married(X,Y).

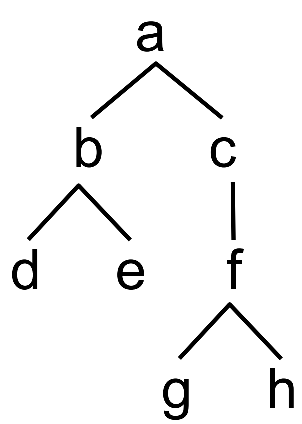
Base case. For simplicity, it says that any X is "above" herself.
If you don't like that, replace base case with above(X,Y) :- boss(X,Y).

Recursive queries

- **above(X,X).**
- above(X,Y) :- boss(X,Underling), above(Underling,Y).

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e). ...

- above(c,h). % should return Yes
  - matches above(X,X)? no



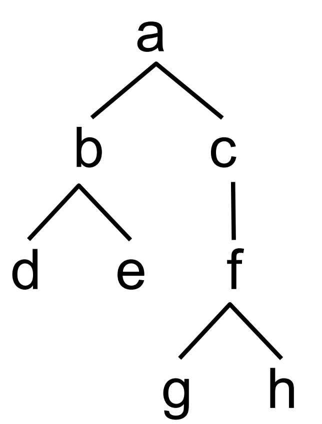
Recursive queries

TECHNISCHE UNIVERSITÄT DARMSTADT

- above(X,X).
- **above(X,Y) :- boss(X,Underling), above(Underling,Y).**

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e). …

- above(c,h). % should return Yes
  - matches above(X,Y) with X=c, Y=h
    - boss(c,Underling),
      - matches boss(c,f) with Underling=f
    - above(f, h).
      - matches above(X,X)? no

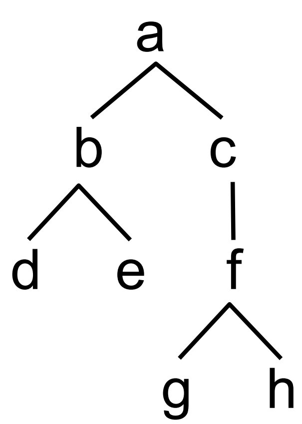


Recursive queries

- above(X,X).
- above(X,Y) :- boss(X,Underling), above(Underling,Y).

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e). …

- above(c,h). % should return Yes
  - matches above(X,Y) with X=c, Y=h
    - boss(c,Underling),
      - matches boss(c,f) with Underling=f
    - above(f, h).
      - matches above(X,Y) with X=f, Y=h
        (local copies of X,Y distinct from previous call)
        - boss(f,Underling),
        - matches boss(f,g) with Underling=g
        - above(g, h).
        - …ultimately fails because g has no underlings …

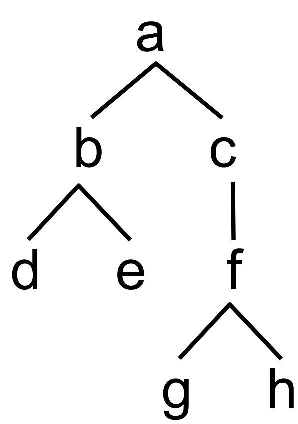


Recursive queries

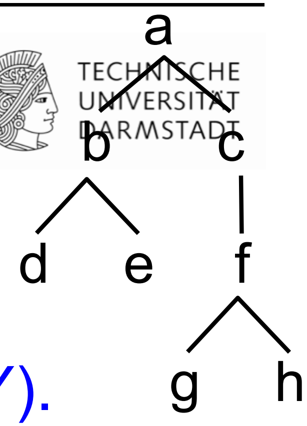
- above(X,X).
- above(X,Y) :- boss(X,Underling), above(Underling,Y).

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e). …

- above(c,h). % should return Yes
  - matches above(X,Y) with X=c, Y=h
    - boss(c,Underling),
      - matches boss(c,f) with Underling=f
    - above(f, h).
      - matches above(X,Y) with X=f, Y=h
        (local copies of X,Y distinct from previous call)
        - boss(f,Underling),
        - matches boss(f,h) with Underling=h
        - above(h, h).
        - matches above(X,X) with X=h



Ordering constraints for speed



- above(X,X).
- above(X,Y) :- boss(X,Underling), above(Underling,Y).

- Which is more efficient?
- above(c,h), friends(c,h).
- friends(c,h), above(c,h).

Probably quicker to check first whether they're friends. If they're not, can skip the whole long above(c,h) computation, which must iterate through descendants of c.

Which is more efficient?

above(X,Y), friends(X,Y).

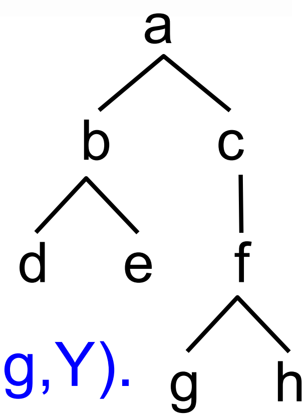
friends(X,Y), above(X,Y).

For each boss X, iterate through all Y below her and check if each Y is her friend.

(Worse to start by iterating through all friendships: if X has 5 friends Y, we scan all the people below her 5 times, looking for each friend in turn.)

Ordering constraints for speed

TECHNISCHE UNIVERSITÄT DARMSTADT



- above(X,X).

- Which is more efficient?

"query modes"
1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- boss(Overling,Y), above(X,Overling).

+,+
- If the query is above(c,e)?

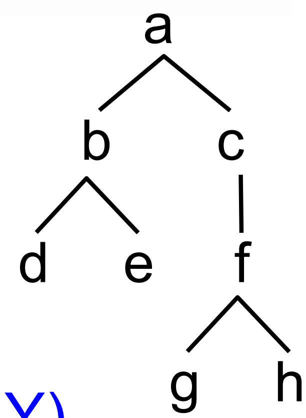
  1. iterates over descendants of c, looking for e
  2. iterates over ancestors of e, looking for c.
  2. is better: no node has very many ancestors, but some have a lot of descendants.

+,-
- If the query is above(c,Y)? 1. is better. Why?

-,+
- If the query is above(X,e)? 2. is better. Why?

-,-
- If the query is above(X,Y)? Doesn't matter much. Why?

Ordering constraints for speed



- above(X,X).

- Which is more efficient?

1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- above(Underling,Y), boss(X,Underling).

2. takes forever – literally!! Infinite recursion.
Here's how:

above(c,h). % should return Yes

  matches above(X,Y) with X=c, Y=h

    above(Underling, h)

      matches above(X,X) with local X = Underling = h

    boss(c, h) (our current instantiation of boss(X, Underling))

      no match

Prolog also allows complex terms

- What we've seen so far is called Datalog: "databases in logic."
- Prolog is "programming in logic." It goes a little bit further by allowing complex terms, including records, lists and trees.
- These complex terms are the source of the only hard thing about Prolog, "unification."

Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- Several essentially identical ways to find older students:
- at_jhu(student(IDNum, Name, date(Day,Month,Year))),
  Year < 1983.
- at_jhu(student(_, Name, date(_,_,Year))),
  Year < 1983.
- at_jhu(Person),
  Person=student(_,_,Birthday),
  Birthday=date(_,_,Year),
  Year < 1983.

usually no need to use =
but sometimes it’s nice
to introduce a temporary name
especially if you’ll use it twice

This query binds Person and Birthday to complex structured values, and Year to an int. Prolog prints them all.

Complex terms

```
at_jhu(student(128327, 'Spammy K',  date(2, may, 1986))).
at_jhu(student(126547, 'Blobby B',  date(15, dec, 1985))).
at_jhu(student(456591, 'Fuzzy W',   date(23, aug, 1966))).

student_get_bday(Stu , Bday) :- Stu=student(_, _, Bday) .
date_get_year(Date,Year) :- Date=date(_, _, Year).
```

bad style

- So you could write accessors in object-oriented style:

```
student_get_bday(Student,Birthday),
    date_get_year(Birthday,Year),
    at_jhu(Student), Year < 1983.
```

- Answer:

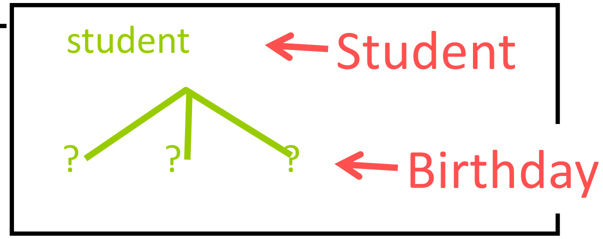
```
    Student=student(456591, 'Fuzzy W', date(23, aug, 1966)),
    Birthday=date(23, aug, 1966),
    Year=1966.
```

Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- ~~student_get_bday(Stu , Bday) :- Stu=student(_, _, Bday) .~~
- ~~date_get_year(Date,Year) :- Date=date(_, _, Year).~~ bad style

- So you could write accessors in object-oriented style:
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.



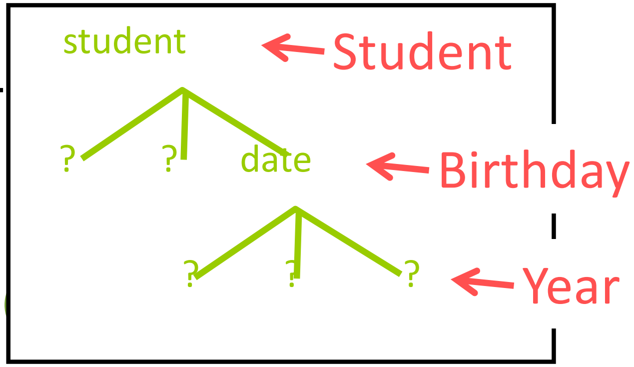
- Answer:
  Student=student(456591, ‘Fuzzy W’, date(23, aug, 1966)),
  Birthday=date(23, aug, 1966),
  Year=1966.

Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- ~~student_get_bday(Stu , Bday) :- Stu=student(_, _, Bday) .~~
- ~~date_get_year(Date,Year) :- Date=date(_, _, Year).~~ bad style

- So you could write accessors in object-oriented style:
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.
- Answer:
  Student=student(456591, ‘Fuzzy W’, date
  Birthday=date(23, aug, 1966),
  Year=1966.

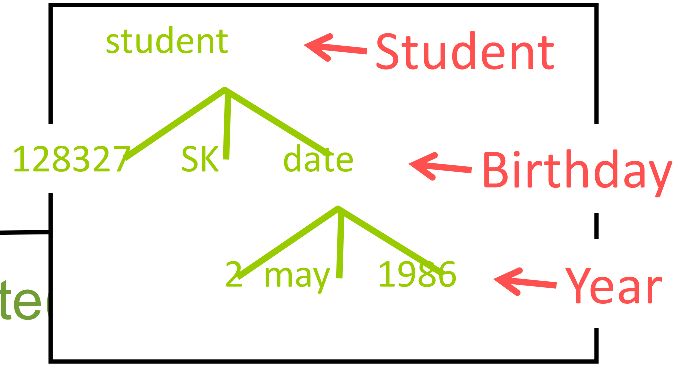


Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- ~~student_get_bday(Stu , Bday) :- Stu=student(_, _, Bday) .~~
- ~~date_get_year(Date,Year) :- Date=date(_, _, Year).~~ bad style

- So you could write accessors in object-oriented style:
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.
- Answer:
  Student=student(456591, ‘Fuzzy W’, date
  Birthday=date(23, aug, 1966),
  Year=1966.



Complex terms

- at_jhu(student(128327, 'Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, 'Blobby B', date(15, dec, 1985))).
- at_jhu(student(456591, 'Fuzzy W', date(23, aug, 1966))).

- ~~student_get_bday(Stu, Bday) :- Stu=student(_, _, Bday).~~
- ~~date_get_year(Date,Year) :- Date=date(_, _, Year).~~ bad style

- So you could write accessors in object
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.
- Answer:
  Student=student(456591, 'Fuzzy W', date
  Birthday=date(23, aug, 1966),
  Year=1966.

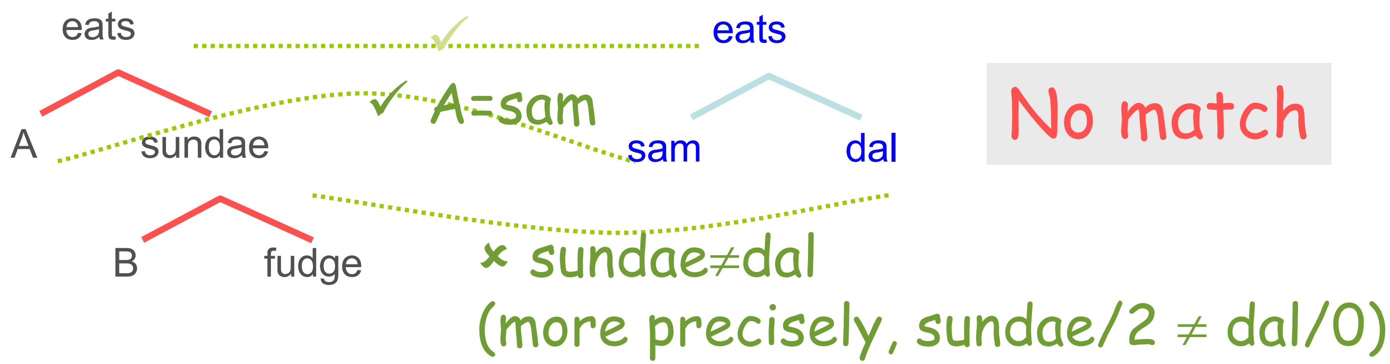
Fail

(and backtrack)

How does matching happen?

- **eats(sam, dal).**
- eats(josie, sundae(vanilla, caramel)).
- eats(rajiv, sundae(mintchip, fudge)).
- eats(robot('C-3PO'), Anything). % variable in a fact

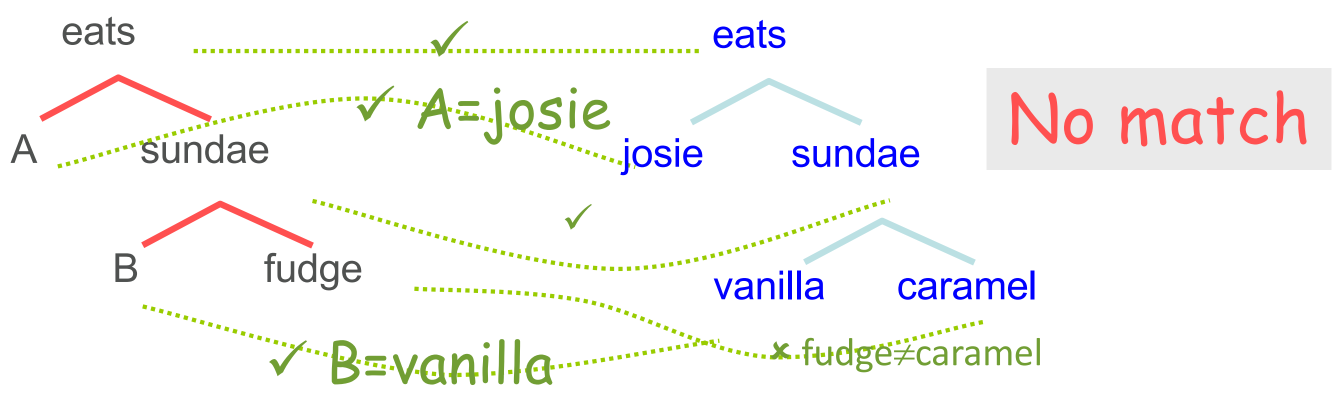
- Query: eats(A, sundae(B,fudge)).
- What happens when we try to match this against facts?



How does matching happen?

- eats(sam, dal).
- **eats(josie, sundae(vanilla, caramel)).**
- eats(rajiv, sundae(mintchip, fudge)).
- eats(robot('C-3PO'), Anything). % variable in a fact

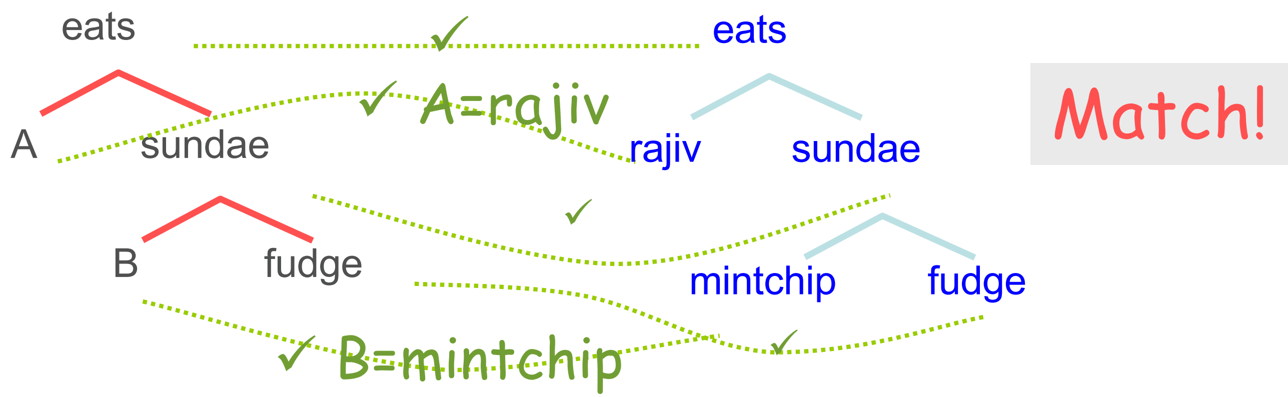
- Query: eats(A, sundae(B,fudge)).
- What happens when we try to match this against facts?



How does matching happen?

- eats(sam, dal).
- eats(josie, sundae(vanilla, caramel)).
- **eats(rajiv, sundae(mintchip, fudge)).**
- eats(robot('C-3PO'), Anything). % variable in a fact

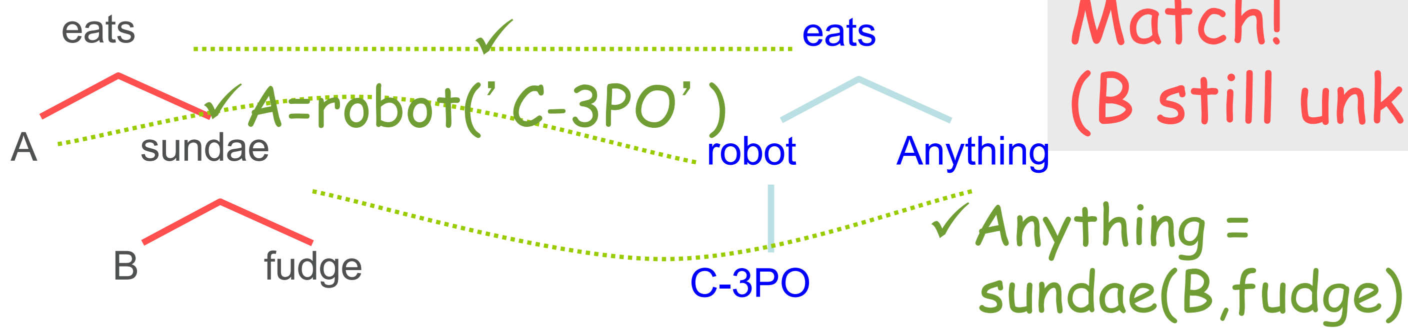
- Query: eats(A, sundae(B,fudge)).
- What happens when we try to match this against facts?



How does matching happen?

- eats(sam, dal).
- eats(josie, sundae(vanilla, caramel)).
- eats(rajiv, sundae(mintchip, fudge)).
- **eats(robot('C-3PO'), Anything).** % variable in a fact

- Query: eats(A, sundae(B,fudge)).
- What happens when we try to match this against facts?



Match!
(B still unknown)

Family trees (just Datalog here) …

```
female(sarah).
female(rebekah).
female(hagar_concubine).
female(milcah).
female(bashemath).
female(mahalath).
female(first_daughter).
female(second_daughter).
female(terahs_first_wife).
female(terahs_second_wife).
female(harans_wife).
female(lots_first_wife).
female(ismaels_wife).
female(leah).
female(kemuels_wife).
female(rachel).
female(labans_wife).
```

```
male(terah).
male(nahor).
male(isaac).
male(uz).
male(bethuel).
male(iscah).
male(jacob).
male(hadad).
male(reuel).
male(judah4th).
male(elak).
male(ben-ammi).
```

```
male(abraham).
male(haran).
male(ismael).
male(kemuel).
male(lot).
male(esau).
male(massa).
male(laban).
male(levi3rd).
male(aliah).
male(moab).
```

Family trees (just Datalog here) …

```
father(terah, sarah).
father(terah, abraham).
father(terah, nahor).
father(terah, haran).
father(abraham, isaac).
father(abraham, ismael).
father(nahor, uz).
father(nahor, kemuel).
father(nahor, bethuel).
father(haran, milcah).
father(haran, lot).
father(haran, iscah).
father(isaac, esau).
father(isaac, jacob).
father(ismael, massa).
father(ismael, mahalath).
father(ismael, hadad).
father(ismael, bashemath).
father(esau, reuel).
father(jacob, levi3rd).
father(jacob, judah4th).
father(esau, aliah).
father(esau, elak).
father(kemuel, aram).
father(bethuel, laban).
father(bethuel, rebekah).
father(lot, first_daughter).
father(lot, second_daughter).
father(lot, moab).
father(lot, ben_ammi).
father(laban, rachel).
father(laban, leah).
```

```
mother(terahs_second_wife, sarah).
mother(terahs_first_wife, abraham).
mother(terahs_first_wife, nahor).
mother(terahs_first_wife, haran).
mother(sarah, isaac).
mother(hagar_concubine, ismael).
mother(milcah, uz).
mother(milcah, kemuel).
mother(milcah, bethuel).
mother(harans_wife, milcha).
mother(harans_wife, lot).
mother(harans_wife, iscah).
mother(rebekah, esau).
mother(rebekah, jacob).
mother(ismaels_wife, massa).
mother(ismaels_wife, mahalath).
mother(ismaels_wife, hadad).
mother(ismaels_wife, bashemath).
mother(bethuels_wife, laban).
mother(bethuels_wife, rebekah).
mother(lots_first_wife, first_daughter).
mother(lots_first_wife, second_daughter).
mother(first_daughter, moab).
mother(second_daughter, ben_ammi).
mother(bashemath, reuel).
mother(leah, levi3rd).
mother(leah, judas4th).
mother(mahalath, aliah).
mother(mahalath, elak).
mother(lebans_wife, rachel).
mother(lebans_wife, leah).
```

Family trees (just Datalog here) …

- husband(terah, terahs_first_wife).
  husband(terah, terahs_second_wife).
  husband(abraham, sarah).
  husband(abraham, hagar_concubine).
  husband(nahor, milcah).
  husband(haran, harans_wife).
  husband(isaac, rebekah).
  husband(ismael, ismaels_wife).
  husband(kemuel, kemuels_wife).
  husband(bethuel, bethuels_wife).
  husband(lot, lots_first_wife).
  husband(lot, first_daughter).
  husband(lot, second_daughter).
  husband(esau, bashemath).
  husband(jacob, leah).
  husband(jacob, rachel).
  husband(esau, mahalath).
  husband(laban, labans_wife).

- wife(X, Y):- husband(Y, X).
- married(X, Y):- wife(X, Y).
- married(X, Y):- husband(X, Y).

Does husband(X,Y) mean
"X is the husband of Y"
or
"The husband of X is Y"?
Conventions vary … pick one and stick to it!

convention in these slides

Family trees (just Datalog here) …

- % database
  mother(sarah,isaac).
  father(abraham,isaac).
  …
- parent(X, Y):- mother(X, Y).
  parent(X, Y):- father(X, Y).

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

- You may refactor this code to avoid duplication:
  - better handling of male/female
    - currently grandmother and grandfather repeat the same “X…Z…Y” pattern
  - better handling of generations
    - currently great_grandmother and great_grandfather would repeat it again

Family trees (just Datalog here) …

- Refactored database (now specifies parent, not mother/father):
  - parent(sarah, isaac). female(sarah).
  - parent(abraham, isaac). male(abraham).

- Refactored ancestry (recursive, gender-neutral):
  - anc(0,X,X).
  - anc(N,X,Y) :- parent(X,Z), anc(N-1,Z,Y).

- Now just need one clause to define each English word:
  - parent(X,Y) :- anc(1,X,Y).
mother(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(X).
  - grandparent(X,Y) :- anc(2,X,Y).
grandmother(X,Y):- grandparent(X,Y), female(X).
grandfather(X,Y) :- grandparent(X,Y), male(X).
  - great_grandparent(X,Y) :- anc(3,X,Y).
etc.

A few more examples of family relations
(only the gender-neutral versions are shown)

- half_sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.
- sibling(X,Y) :- mother(Z,X), mother(Z,Y), father(W,X), father(W,Y), X \=Y.
  - Warning: This inequality constraint X \= Y only works right in mode +,+.
  - (It asks whether unification *would fail.* So the answer to A \= 4 is “no”, since A=4 would succeed! There is no way for Prolog to represent that A can be “anything but 4” – there is no “anything but 4” term.)
- aunt_or_uncle(X,Y) :- sibling(X,Z), parent(Z,Y).
- cousin(X,Y):- parent(Z,X), sibling(Z,W), parent(W,Y).
- deepcousin(X,Y):- sibling(X,Y). % siblings are 0^{th} cousins
- deepcousin(X,Y):- parent(Z,X), deepcousin(Z,W), parent(W,Y).

  % we are Nth cousins if we have parents who are (N-1)st cousins

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]

- if $X=\underbrace{[3,4]}_{\text{cons(3,cons(4,nil))}}$, then $\underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}}=\overbrace{[1,2,3,4]}^{\text{cons(1, cons(2, cons(3, cons(4, nil))))}}$

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]

- $\overbrace{[1,2,3,4]}^{\text{cons(1, cons(2, cons(3, cons(4, nil))))}}=\underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}} \rightarrow \underbrace{X=[3,4]}_{\text{cons(3,cons(4,nil))}}$ by unification

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]

- $\overbrace{[1,2]}^{\text{cons(1, cons(2, nil))}} \quad = \underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}} \rightarrow \underbrace{X=[]}_{\text{nil}}$

Decomposing lists

- first(X,List) :- ...?

- first(X,List) :- List=[X|Xs].
  - Traditional variable name:
    "X followed by some more X's."
- first(X, [X|Xs]).
  - Nicer: eliminates the single-use variable List.
- first(X, [X|_]).
  - Also eliminate the single-use variable Xs.

List processing: member

- member(X,Y) should be true if X is any object, Y is a list, and X is a member of the list Y.

- member(X, [X|_]).   % same as "first"
- member(X, [Y|Ys]) :- member(X,Ys).
- Query: member(giraffe, [beaver, ant, steak(giraffe), fish]).
  - Answer: no (why?)

List processing: member

- Query: member(X, [7,8,7]).
  - Answer: X=7 ;
    X=8 ;
    X=7
- Query: member(7, List).
  - Answer: List=[7 | Xs] ;
    List=[X1, 7| Xs] ;
    List=[X1, X2, 7 | Xs] ;
    … (willing to backtrack forever)

Arithmetic in <u>pure</u> Prolog

TECHNISCHE UNIVERSITÄT DARMSTADT

- Let's rethink arithmetic as term unification!
- Let us divide 6 by 2
  by making Prolog prove that $\exists x\ 2*x = 6$.
- Query: times(2,X,6). So how do we program
  times?
  Represent 0 by z (for "zero")
  Represent 1 by s(z) (for "successor").
  Represent 2 by s(s(z))
  Represent 3 by s(s(s(z)))
  … "Peano integers"

- So actually our query times(2,X,6) will be written
  times(s(s(z)), X, s(s(s(s(s(s(z))))))).

A pure Prolog definition of length

- length([ ],z).
- length([_|Xs], s(N)) :- length(Xs,N).

- This is pure Prolog and will work perfectly everywhere.
- Yeah, it's a bit annoying to use Peano integers for input/output:
  - Query: length([[a,b],[c,d],[e,f]], N).
    Answer: N=s(s(s(z)))
  - Query: length(List, s(s(s(z)))).
    Answer: List=[A,B,C]

yuck?

- But you could use impure Prolog to convert them to "ordinary" numbers just at input and output time …

A pure Prolog definition of length

- length([ ],z).
- length([_|Xs], s(N)) :- length(Xs,N).

- This is pure Prolog and will work perfectly everywhere.
- Converting between Peano integers and ordinary numbers:
  - Query: length([[a,b],[c,d],[e,f]], N), decode(N,D).
    Answer: N=s(s(s(z))), D=3
  - Query: encode(3,N), length(List, N).
    Answer: N=s(s(s(z))), List=[A,B,C]

- Using Prolog's built-in arithmetic:
  - decode(z,0). decode(s(N),D) :- decode(N,E), D is E+1.
  - encode(0,z). encode(D,s(N)) :- D > 0, E is D-1, encode(E,N).

Declarative sorting; this is not efficient

- ordered([]).
- ordered([X]).
- ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).

A bad SAT solver (no short-circuit evaluation or propagation)

// Suppose formula uses 5 variables: A, B, C, D, E

- for $A \in \{0, 1\}$
  - for $B \in \{0, 1\}$
    - for $C \in \{0, 1\}$
      - for $D \in \{0, 1\}$
        - for $E \in \{0, 1\}$

          if formula is true

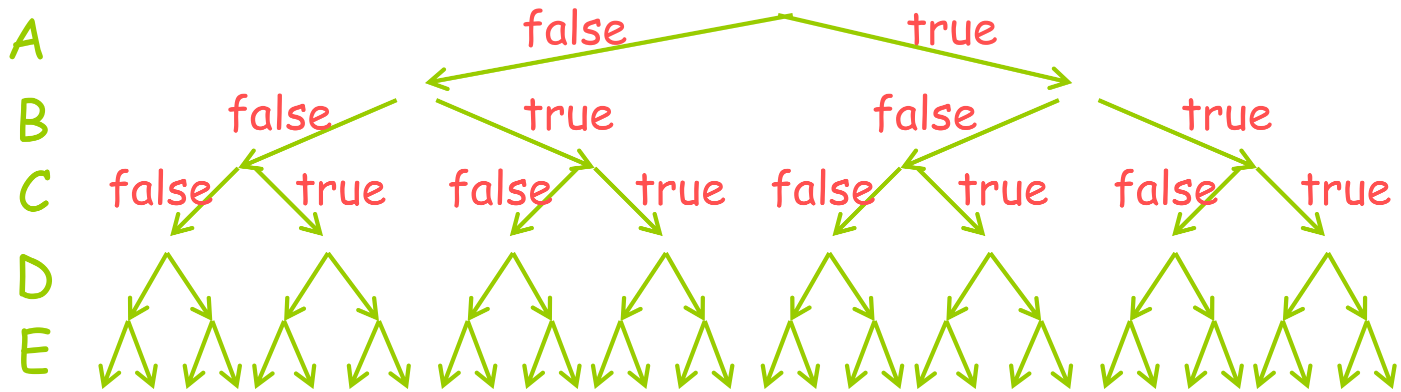
            immediately return (A,B,C,D,E)
- return UNSAT

A bad SAT solver in Prolog

- Query (what variable & value ordering are used here?)
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - % formula (A v ~C v D) ^ (~B v C v E) ^ (A xor E) ^ …
  - formula(A,B,C,D,E) :-
    clause1(A,C,D), clause2(B,C,E), xor(A,E), …
  - % clauses in that formula
  - clause1(true,_,_). clause1(_,false,_). clause1(_,_,true).
  - clause2(false,_,). clause2(_,true,_). clause2(_,_,true).
  - xor(true,false). xor(false,true).

A bad SAT solver in Prolog

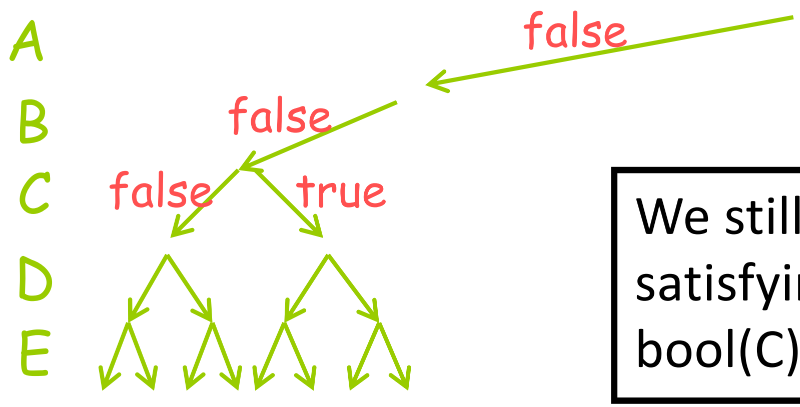
- Query (what variable & value ordering are used here?)
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).



The Prolog cut operator, "!"

- Query
  - bool(A),bool(B),!, bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for
  - bool(false). bool(true).
  - …

Cuts off part of the search space.
Once we have managed to satisfy bool(A),bool(B) and gotten past !, we are committed to our choices so far and won't backtrack to revisit them.

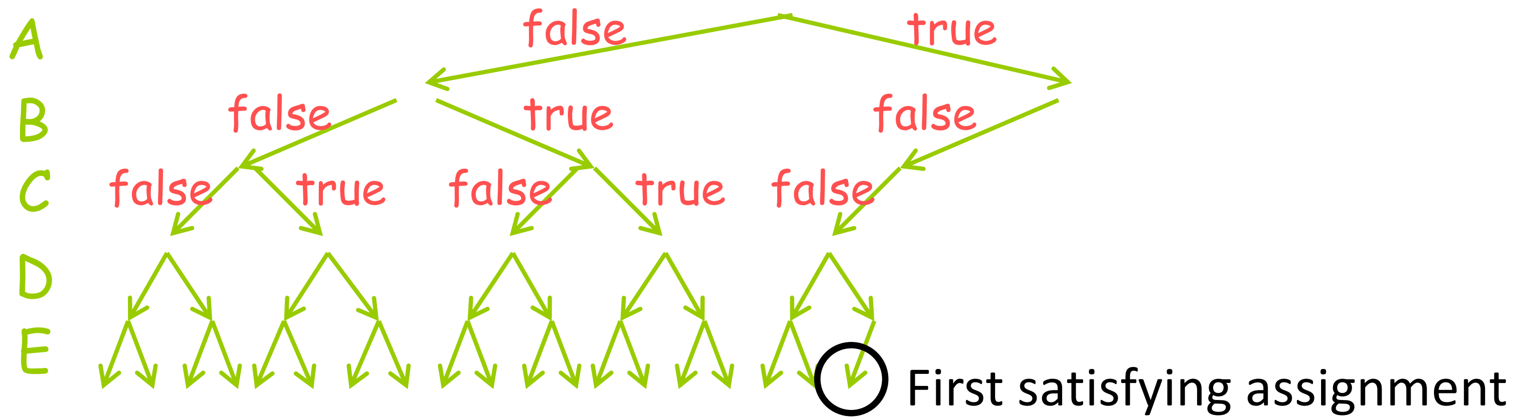


We still backtrack to find other ways of satisfying the subsequent constraints bool(C),bool(D),…

The Prolog cut operator, "!"

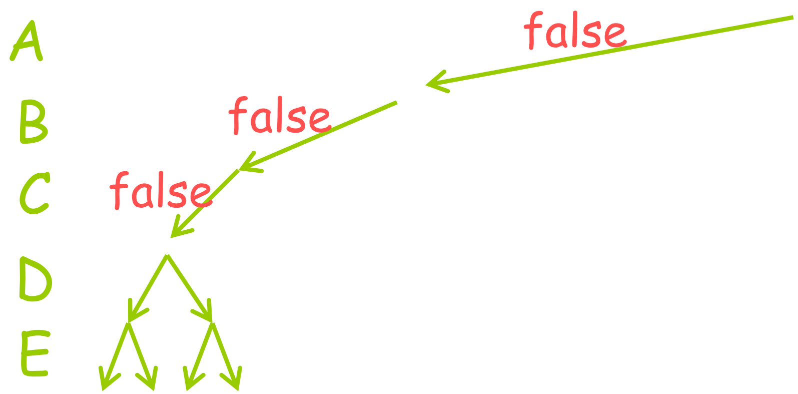
- Query
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E),**!** .
- Program
  - % values available for b
  - bool(false). bool(true).
  - …

Cuts off part of the search space.
Once we have managed to satisfy the constraints before ! (all constraints in this case), we don't backtrack. So we return only first satisfying assignment.



The Prolog cut operator, "!"

- Query
  - bool(A),bool(B),bool(C),**!** ,bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - …

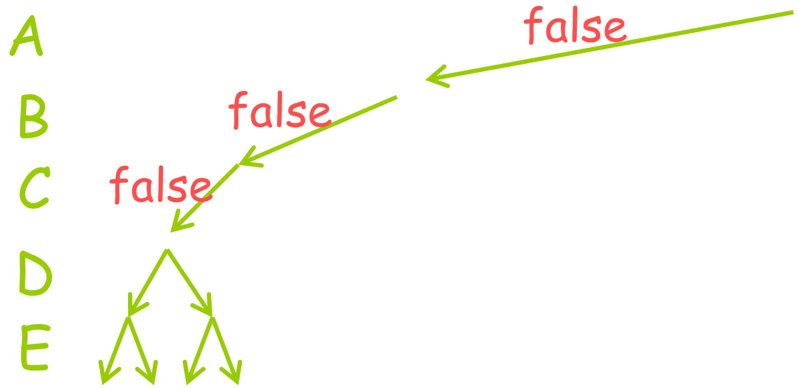


The Prolog cut operator, "!"

- Query
  - bool(A), bool2(B,C), ! bool(D) bool(E) formula(A,B,C,D,E)

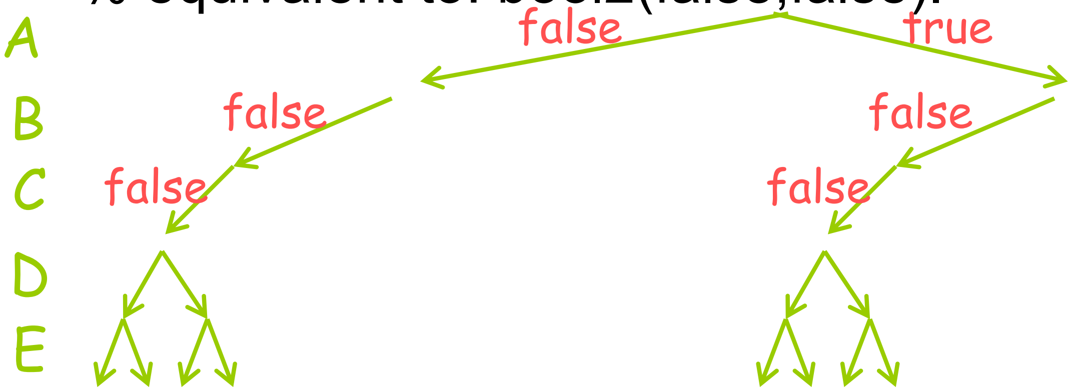
Same effect, using a subroutine.

- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - bool2(X,Y) :- bool(X), bool(Y).



The Prolog cut operator, "!"

- Query
  - bool(A),   bool2(B,C),   ,bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking searc
  - bool(false). bool(true).
  - bool2(X,Y) :- bool(X), bool(Y), ! .
    - % equivalent to: bool2(false,false).



Now effect of "!" is local to bool2. bool2 will commit to its first solution, namely (false,false), not backtracking to get other solutions. But that's just how bool2 works inside. Red query doesn't know bool2 contains a cut; it backtracks to try different A, calling bool2 for each.

What have we learnt?

- Logic is "tool" of choice in AI for representing knoweldge
- Propositional logic cannot speak of objects and relations
- First-oder logic allows to encode true generalities
- Prolog is programming language inspired by logic