CPS721, Sections 051 to 091
Lecture 1

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Stanford Artificial Intelligence course (CS221) and Hector Levesque's textbook and accompanying slides.**

Agenda

- Course Logistics
- History of AI
- Overview of AI
- Introduction to symbolic AI

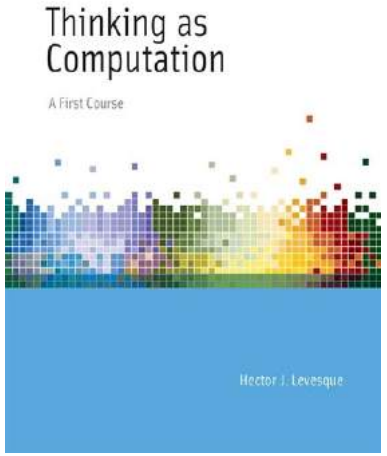
Course Logistics (1)

- Please see the course management form: https://www.cs.ryerson.ca/~mes/courses/cps721/info.html

Section	Status	Day	Start Time	End Time	Room
Lectures	Sections 1-4 (Prof. Soutchanski)	Wednesday	12:00	14:00	zoom
		Friday	12:00	13:00	zoom
Lectures	Sections 5-8 (Prof. Farsad)	Tuesday	08:00	10:00	zoom
		Wednesday	17:00	18:00	zoom
Section 1	Lab/Tutorial	Monday	15:00	16:00	Zoom
Section 2	Lab/Tutorial	Tuesday	17:00	18:00	Zoom
Section 3	Lab/Tutorial	Tuesday	11:00	12:00	Zoom
Section 4	Lab/Tutorial	Thursday	16:00	17:00	Zoom
Section 5	Lab/Tutorial	Tuesday	12:00	13:00	Zoom
Section 6	Lab/Tutorial	Thursday	10:00	11:00	Zoom
Section 7	Lab/Tutorial	Monday	15:00	16:00	Zoom
Section 8 / 9	Lab/Tutorial	Friday	11:00	12:00	Zoom

Course Logistics (2)

- **Compulsory Text Book:**
  Thinking as Computation: A First Course, by *Hector Levesque*. ISBN: 9780262534741.



- **Grading**:

Course Work	Due Date	Grade Value (%)
Assignment 1	September 27, Monday	2
Assignment 2	October 11, Monday	2
Assignment 3	October 25, Monday	2
Midterm	Friday, October 29, 4-6pm EST	20
Assignment 4	November 15, Monday	2
Assignment 5	November 29, Monday	2
Tutorials/Labs	As per timetable above, from Mon, Sept 13	20
Final Exam	TBA	50
Total		100

Agenda

- Course Logistics
- History of AI
- Overview of AI
- Introduction to symbolic AI

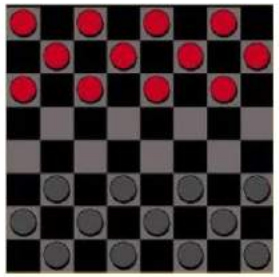
Birth of the AI

1956: Workshop at Dartmouth College; attendees: John McCarthy, Marvin Minsky, Claude Shannon, etc.

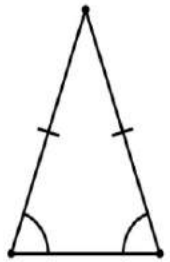
Aim for **general principles**:

Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it.

Birth of AI: Early Successes



Checkers (1952): Samuel's program learned weights and played at strong amateur level



Problem solving (1955): Newell & Simon's Logic Theorist: prove theorems in Principia Mathematica using search + heuristics; later, General Problem Solver (GPS)

Overwhelming optimism...

- Machines will be capable, within twenty years, of doing any work a man can do. —Herbert Simon

- Within 10 years the problems of artificial intelligence will be substantially solved. —Marvin Minsky

- I visualize a time when we will be to robots what dogs are to humans, and I’m rooting for the machines. —Claude Shannon

Overwhelming optimism...

- Machines will be capable, within twenty years, of doing any work a man can do. —Herbert Simon

- Within 10 years the problems of artificial intelligence will be substantially solved. —Marvin Minsky

- I visualize a time when we will be to robots what dogs are to humans, and I’m rooting for the machines. —Claude Shannon

...underwhelming results

Example: machine translation

The spirit is willing but the flesh is weak.

↓

(Russian)

↓

The vodka is good but the meat is rotten.

1966: ALPAC report cut off government funding for MT, first AI winter

What went wrong?

Problems:

- **Limited computation**: search space grew exponentially, outpacing hardware ($100! \approx 10^{157} > 10^{80}$)
- **Limited information**: complexity of AI problems (number of words, objects, concepts in the world)

Contributions:

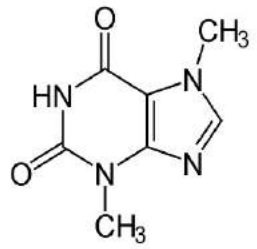
- Lisp, garbage collection, time-sharing (John McCarthy)
- Key paradigm: separate **modeling** and **inference**

Knowledge-based systems (70-80s)

Expert systems: elicit specific domain knowledge from experts in form of rules:

```
if [premises] then [conclusion]
```

Knowledge-based systems (70-80s)



DENDRAL: infer molecular structure from mass spectrometry

MYCIN: diagnose blood infections, recommend antibiotics

XCON: convert customer orders into parts specification;
save DEC $40 million a year by 1986

Knowledge-based systems

Contributions:

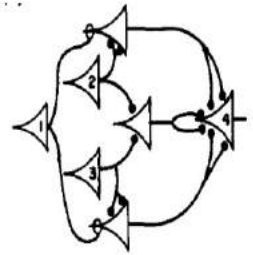
- First **real application** that impacted industry
- Knowledge helped curb the exponential growth

Problems:

- Knowledge is not deterministic rules, need to model **uncertainty**
- Requires considerable **manual effort** to create rules, hard to maintain

1987: Collapse of Lisp machines and second AI winter

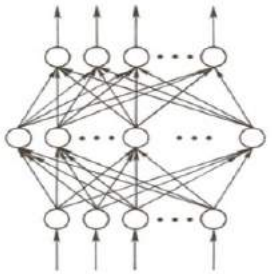
Artificial neural networks



1943: introduced artificial neural networks, connect neural circuitry and logic (McCulloch/Pitts)

1969: Perceptrons book showed that linear models could not solve XOR, killed neural nets research (Minsky/Papert)

Training networks



1986: popularization of backpropagation for training multi-layer networks (Rumelhardt, Hinton, Williams)

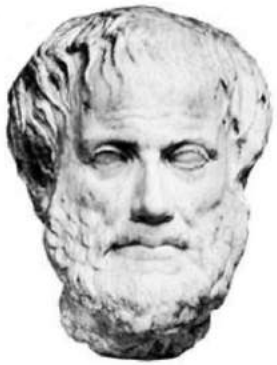
1989: applied convolutional neural networks to recognizing handwritten digits for USPS (LeCun)

Deep learning

AlexNet (2012): huge gains in object recognition; transformed computer vision community overnight

AlphaGo (2016): deep reinforcement learning, defeat world champion Lee Sedol

Two intellectual traditions



- AI has always swung back and forth between the two intellectual traditions
  - One rooted in logic (a.k.a., symbolic AI, knowledge-based system)
  - One rooted in neuroscience (at least initially). Now known as machine learning, deep learning or statistical AI
  - This debate is paralleled in cognitive science with connectionism and computationalism

- Deep philosophical differences, but deeper connections.
  - McCulloch and Pitts' work from 1943 can be viewed as the root of deep learning, but that paper is mostly about how to implement logical operations.
  - The game of Go (and indeed, many games) can be characterized by logic rules. But, most successful systems (AlphaGo) do not tackle the problem directly using logic

Where are we now?

AI agents: how can we create intelligence?

AI tools: how can we benefit society?

An intelligent agent

Perception Robotics Language

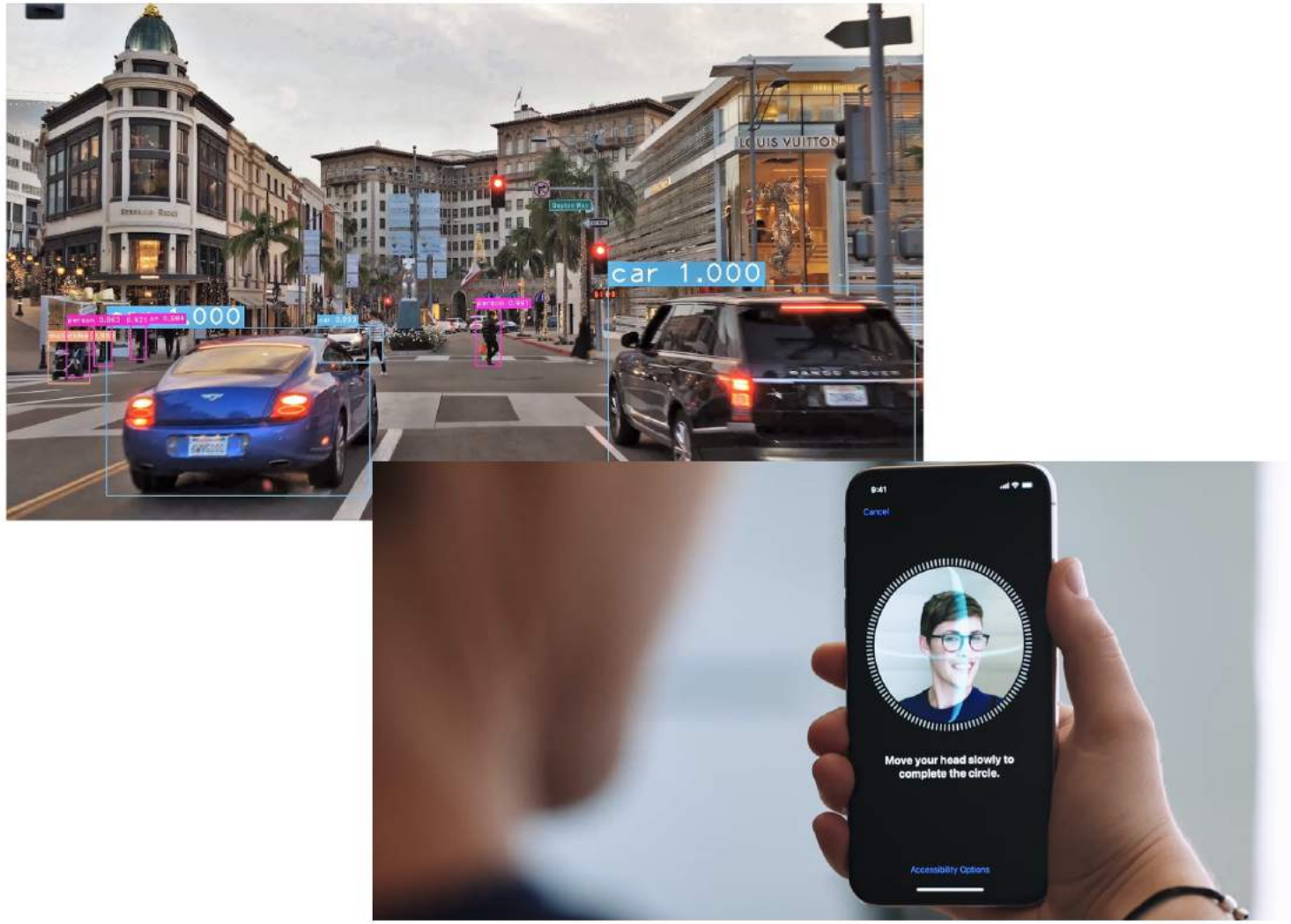
Knowledge Reasoning Learning

Are we there yet?

Machines: narrow tasks, millions of examples

Humans: diverse tasks, very few examples

AI as a tool

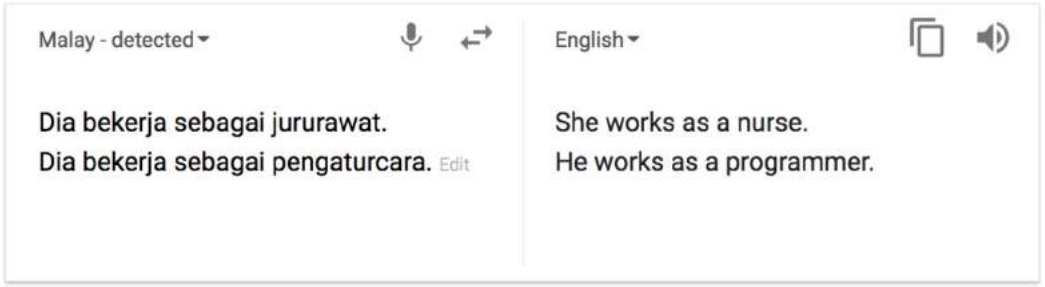

LOUIS VUITTON
car 1.000
Cancel
Move your head slowly to complete the circle.
Accessibility Options

Prospects and risks of AI as tool

AI technology is an amplifier

🙂 Can reduce accessibility barriers and improve efficiency

☹️ Can amplify bias, security risks, centralize power



Summary so far

- AI agents: achieving human-level intelligence, still very far (e.g., generalize from few examples)

- AI tools: need to think carefully about real-world consequences (e.g., security, biases)

Summary so far

- AI agents: achieving human-level intelligence, still very far (e.g., generalize from few examples)

- AI tools: need to think carefully about real-world consequences (e.g., security, biases)

Agenda

- Course Logistics
- History of AI
- Overview of AI
- Introduction to symbolic AI

How do we solve AI tasks?

- The real world is complicated
- We need to write some code (and possibly build some hardware, too)
- This is sometimes referred to as a procedure
- But there is a huge chasm

Modeling-Inference-Learning Paradigm

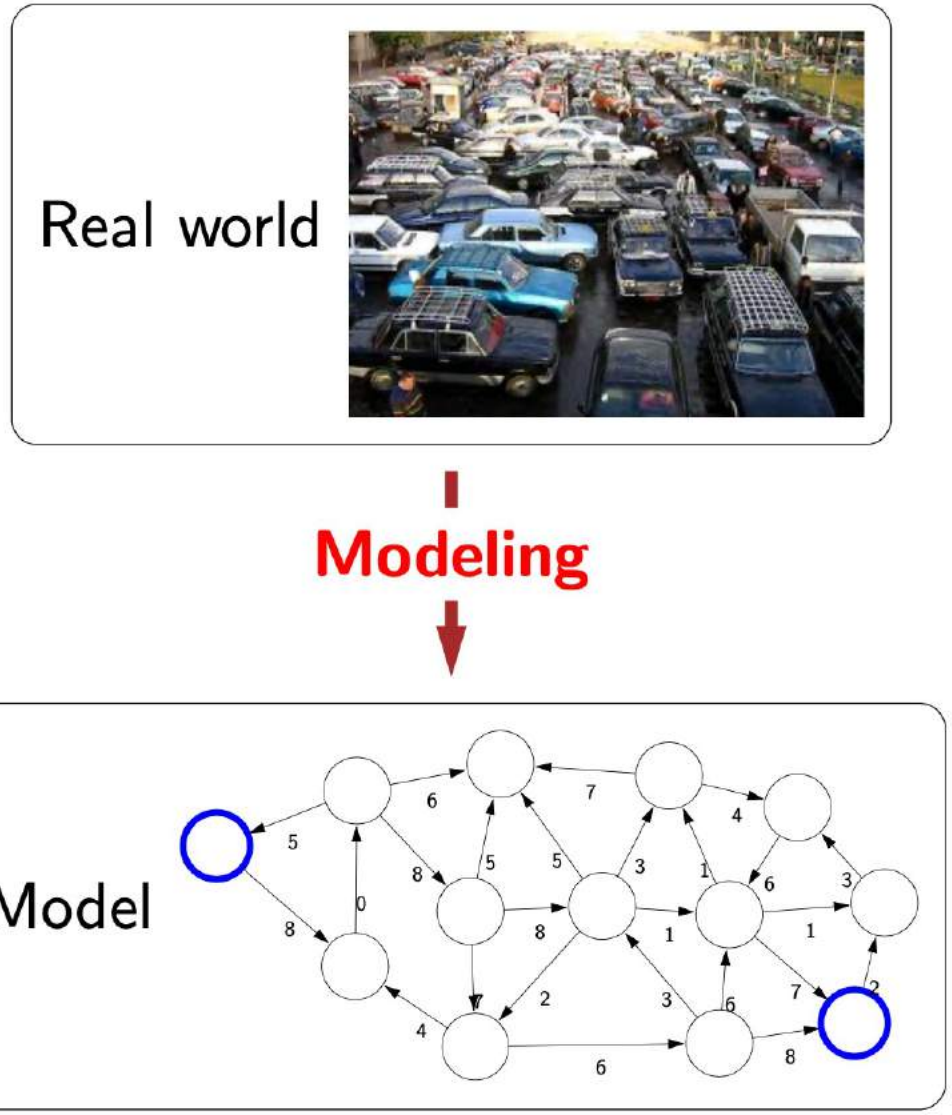
Modeling

Inference

Learning

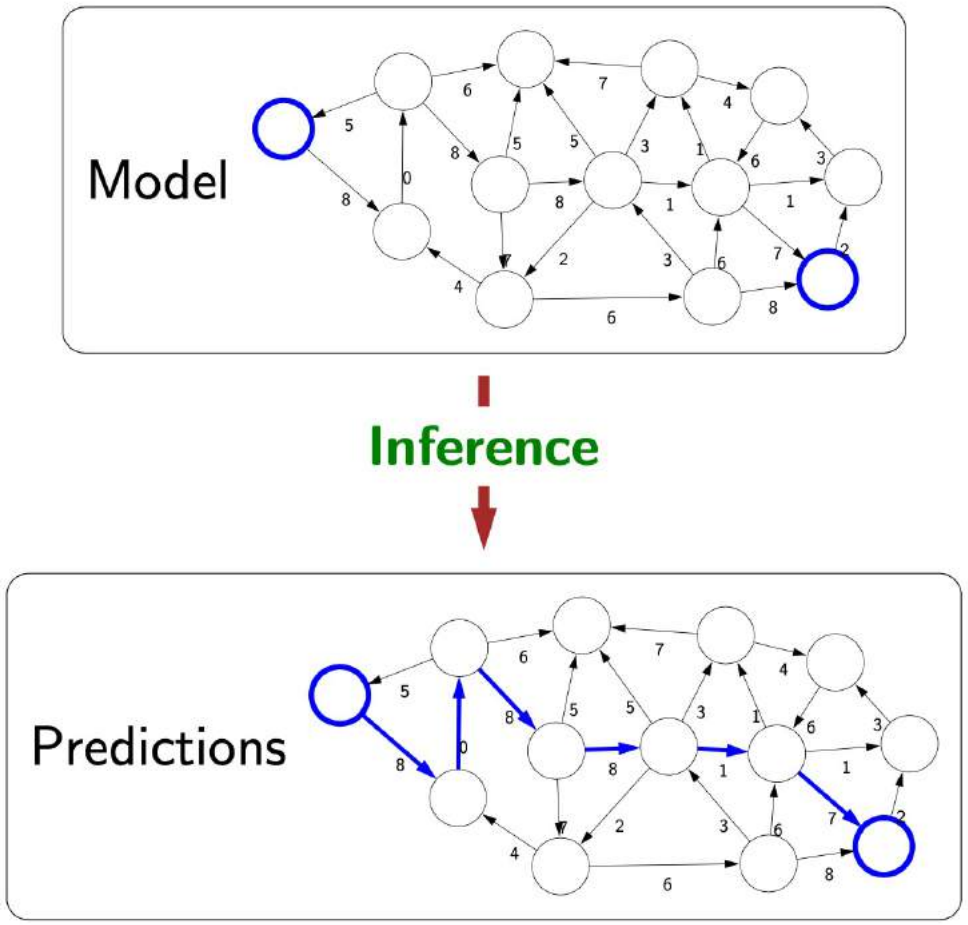
Modeling

- Takes messy real-world problems and packages them into formal mathematical objects
- Models can be subject to rigorous analysis
- Can be operated on by computers
- Modeling is lossy: not all of the richness of the real world can be captured
- What does one keep versus ignore? (An exception to this are games such as Chess, Go or Sodoku, where the real world is identical to the model.)



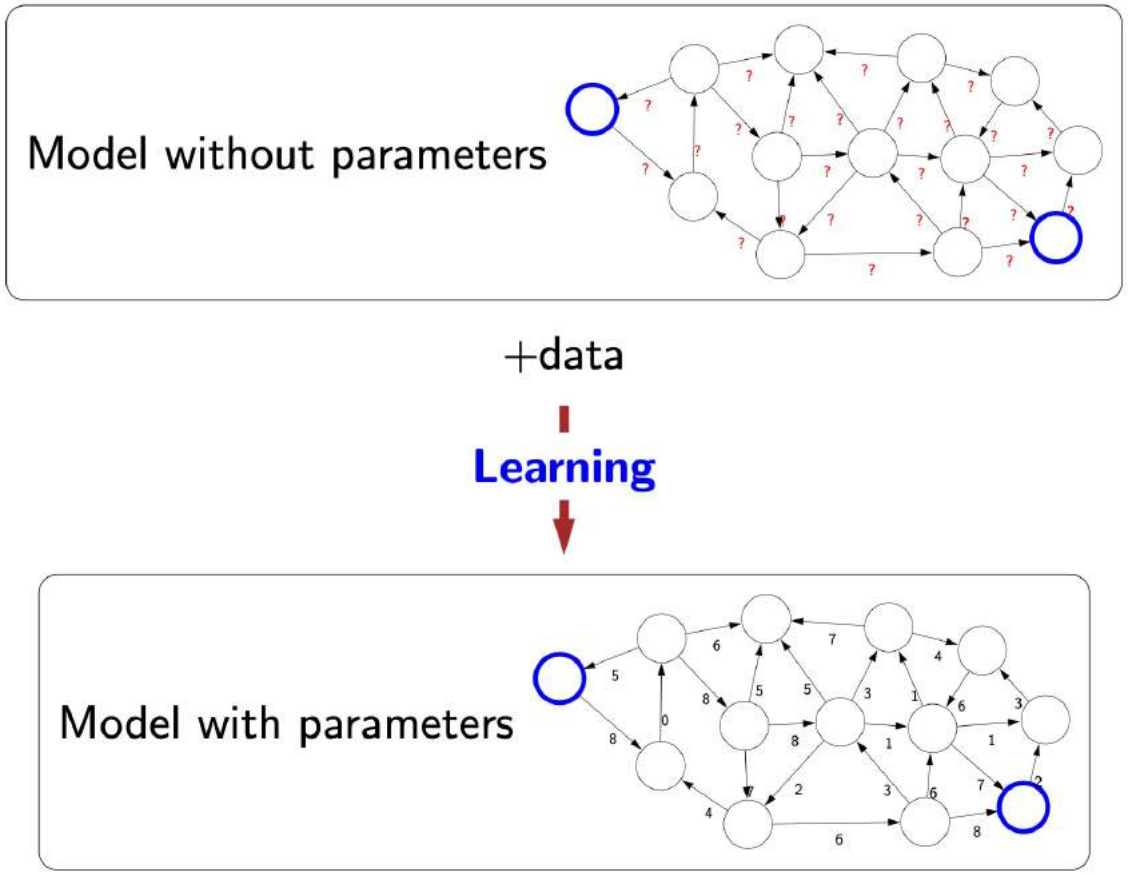
Inference

- Given a model, answer questions with respect to the model
- The focus of inference is usually on efficient algorithms that can answer these questions.
- For some models, computational complexity can be a concern (games such as Go)
- One solution is to use approximations



Learning

- Real world is rich, good models have to be rich as well
- Can’t write a complicated model manually
- (machine) learning gets the model directly from data
- Instead of constructing a model, construct a skeleton of a model (more precisely, a model family)
- Using data data learn the parameters of the model
- Learning here is not tied to a particular approach (e.g., neural networks), but more of a philosophy
- This general paradigm allows one to bridge the gap between logic-based AI and statistical AI



Overview of AI

- Machine learning is the main driver of recent successes in AI as a tool
- Move from “code” to “data” to manage the information complexity
- Requires a leap of faith: generalization

Reflex-based models

Reflex

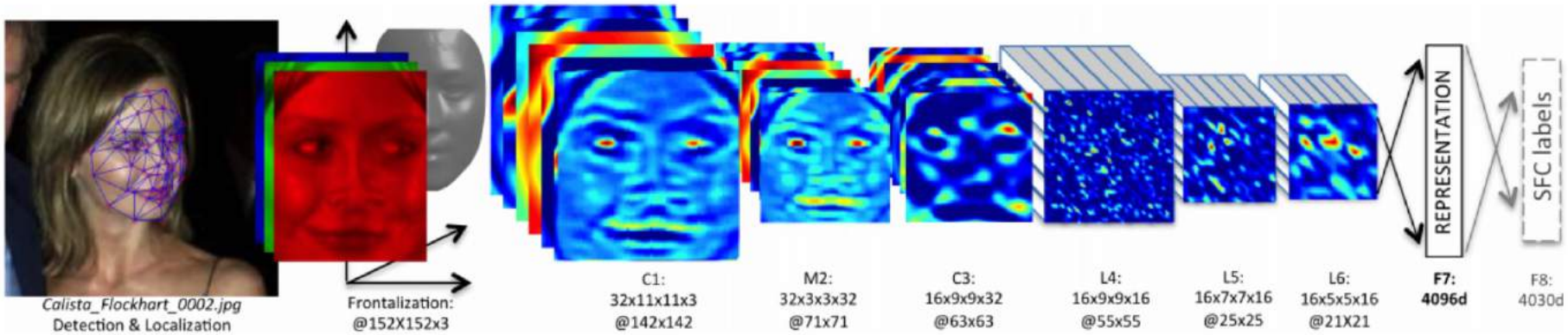
"Low-level intelligence" → "High-level intelligence"

Machine learning

What is this animal?

Reflex-based models

- Examples: linear classifiers, deep neural networks



- Most common models in machine learning

- Fully feed-forward (no backtracking)

State-based models

Search problems

Markov decision processes

Adversarial games

Reflex **States**

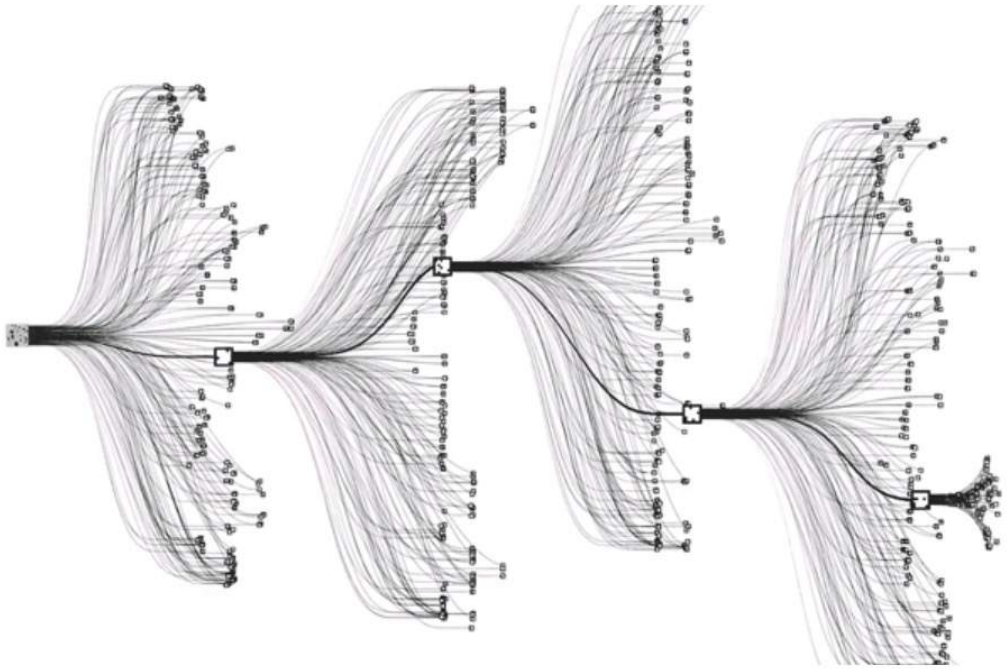
"Low-level intelligence" "High-level intelligence"

Machine learning

What is the best move for white?

State-based models

- Reflex-based models are too simple for tasks that require more forethought (e.g., in playing chess or planning a big trip)
- Model the state of a world and transitions between states which are triggered by actions
- One can think of states as nodes in a graph and transitions as edges
- A lot of efficient algorithms for operating on graphs



Variable-based models

Search problems

Markov decision processes

Adversarial games

Constraint satisfaction problems

Bayesian networks

Reflex **States** **Variables**

"Low-level intelligence" "High-level intelligence"

Machine learning

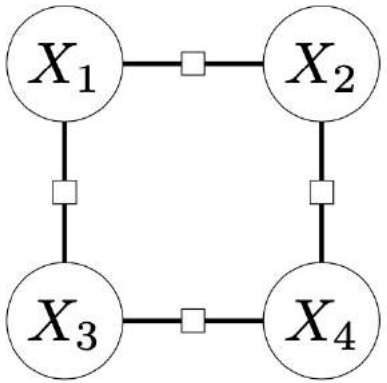
- State-based models are procedural:
  - Specifies step by step instructions on how to go from A to B
- In many applications, the order in which things are done isn't important

Sudoku

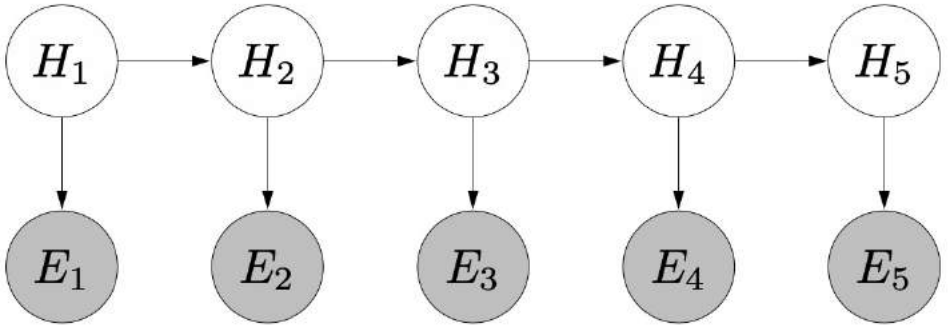
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Variable-based models

Constraint satisfaction problems: hard constraints (e.g., Sudoku, scheduling)



Bayesian networks: soft dependencies (e.g., tracking cars from sensors)



Logic

Search problems

Markov decision processes

Adversarial games

Constraint satisfaction problems

Bayesian networks

Reflex **States** **Variables** **Logic**

"Low-level intelligence" "High-level intelligence"

Machine learning

Virtual Assistant

Tell information

Ask questions

- Digest heterogenous information
- Reason deeply with that information

Logic

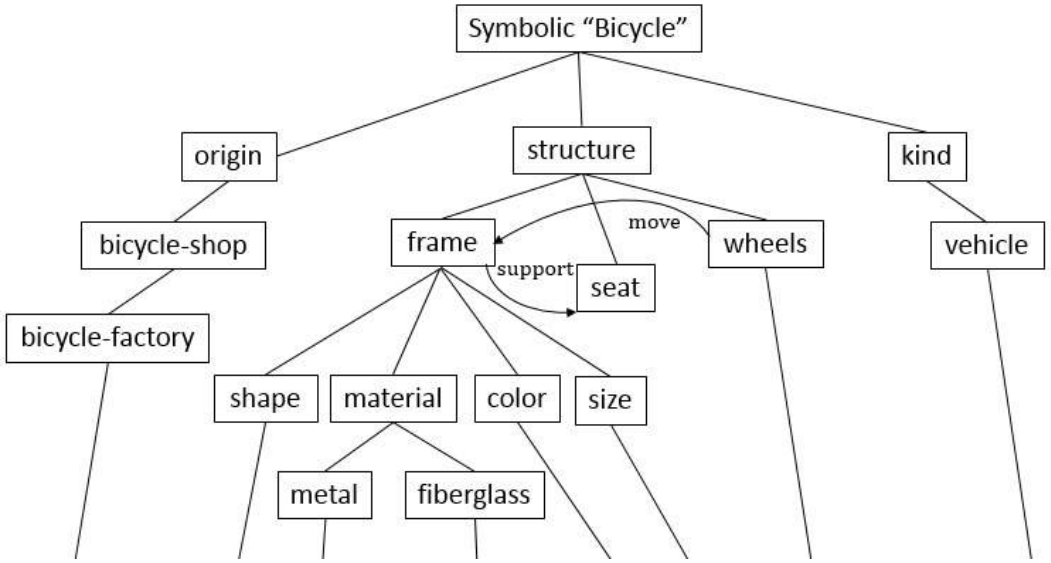
- Logic provides a compact language for modeling which gives us more expressivity

- Historically, logic was one of the first things that AI researchers started with in the 1950s

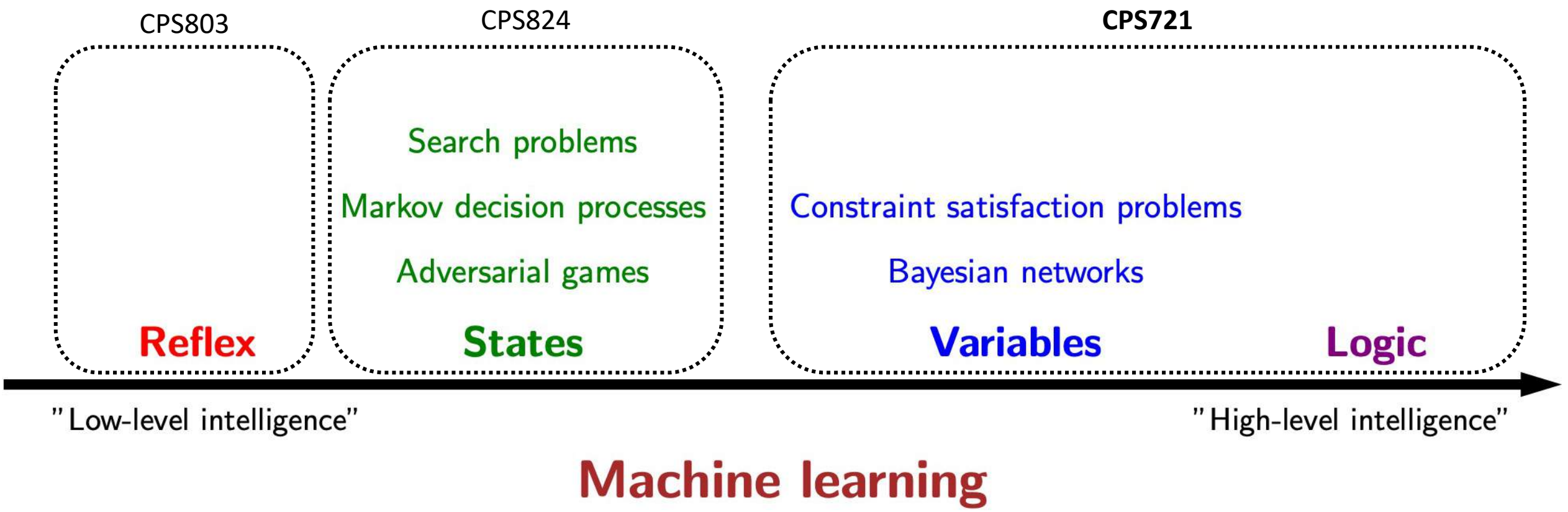
- While logical approaches are quite sophisticated, they did not work well on complex real-world tasks with noise and uncertainty.

- Methods based on probability and machine learning naturally handle noise and uncertainty
  - This is why they presently dominate the AI landscape.
  - But, they are yet to be applied successfully to tasks that require really sophisticated reasoning.

- An active area of research is to combine the richness of logic with the robustness and agility of machine learning



In this course



Agenda

- Course Logistics
- History of AI
- Overview of AI
- Introduction to symbolic AI

Computer Science

- It's not about *computers*!
- Having problems with your PC?
  - Don't ask a Computer Scientist!
  - The electronic / physical properties of computers play little or no role in Computer Science
  - Another analogy to think about: musical instruments *vs.* music Like music, Computer Science is not about anything physical!
- Computer Science is about *computation*:
  - a certain form of manipulation of *symbols*
- Modern electronic computers (like an *Apple iMac* or a *Dell PC*) just happen to provide a fast, cheap, and reliable medium for computation.

Symbols and symbolic structures

- Simplest sort: characters from an alphabet
  - digits: 3, 7, V
  - letters: a, R, α
  - operators: +, $\times$, $\cap$
- String together into more complex ones
  - numerals: 335.42
  - words: “don’t”
- More complex groupings
  - expressions: $247 + 4(x - 1)^3$
  - phrases: “the woman John loved”
- Into truth-valued symbolic structures
  - relations: $247 + 4(x - 1)^3 > \frac{z!}{5}$
  - sentences: “The woman John loved had brown hair.”

Manipulating symbols

The idea: take strings of symbols, break them apart, compare them, and reassemble them according to a recipe called a *procedure.*

It is important to keep track of where you are, and follow the instructions in the procedure exactly. (You may not be able to figure *why* you are doing the steps involved!)

The symbols you have at the start are called the *inputs*. Some of the symbols you end up with will be designated as the *outputs*.

> We say that the procedure is *called* on the inputs and *returns* or produces the outputs.

Then construct more complex procedures out of simple procedures.

In the next few slides, we will look at an interesting special case: **arithmetic**!

Lesson: arithmetic as symbol manipulation

Arithmetic can be built from more primitive symbol manipulations

- starting with very simple operations (such as table lookup), the ability to string and unstring symbols, compare them *etc.*, we can build procedures to do addition
- using these we can build ever more complex procedures, to do multiplication, division, testing for prime numbers, etc.
- using pairs of numbers we can deal with fractions (rational numbers); and using numbers arranged into matrices, we can solve systems of equations, perform numerical simulations, etc.

As we will see in this course, there are also many interesting cases of symbol manipulation that are not numeric!

A key observation

You don't need to know what you're doing!

> To get meaningful answers, you do not have to understand what the symbols stand for, or why the manipulations are *correct*.

The symbols can be manipulated completely mechanically and still end up producing significant and interesting results.

This is the trick of computation:

> We can get computers to perform a wide variety of very impressive activities precisely because we are able to describe those activities as a type of symbol manipulation that can be carried out purely mechanically.

This "trick" has turned out to be one of the major inventions of the 20th century. And note: It has nothing to do with electronics!

A key observation

You don't need to know what you're doing!

> To get meaningful answers, you do not have to understand what the symbols stand for, or why the manipulations are *correct*.

The symbols can be manipulated completely mechanically and still end up producing significant and interesting results.

This is the trick of computation:

> We can get computers to perform a wide variety of very impressive activities precisely because we are able to describe those activities as a type of symbol manipulation that can be carried out purely mechanically.

This "trick" has turned out to be one of the major inventions of the 20th century. And note: It has nothing to do with electronics!

A key observation

You don't need to know what you're doing!

> To get meaningful answers, you do not have to understand what the symbols stand for, or why the manipulations are *correct*.

The symbols can be manipulated completely mechanically and still end up producing significant and interesting results.

This is the trick of computation:

> We can get computers to perform a wide variety of very impressive activities precisely because we are able to describe those activities as a type of symbol manipulation that can be carried out purely mechanically.

This "trick" has turned out to be one of the major inventions of the 20th century. And note: It has nothing to do with electronics!

But what does this have to do with thinking?

Let us return to what we called the Key Conjecture from earlier:

> **Key Conjecture:** thinking can be usefully understood as a *computational* process

What does this controversial conjecture amount to?

- **not** that the brain is something like an electronic computer (which it is in some ways, but in very many other ways is not)
- but that the process of *thinking* can be usefully understood as a form of *symbol processing* that can be carried out purely *mechanically* without having to know what you are doing.

This is what we will explore in this course!

While the textbook promotes this conjecture, this is just a conjecture. In modern AI, the consensus is that useful higher-level intelligence might be at the intersection of symbolic AI and statistical AI.

An example

Consider the following:

I know that my keys are either in my coat pocket or on the fridge.
That's where I always leave them.

I feel my coat pocket and I see that there is nothing there.

So my keys must be on the fridge.
And that's where I should go looking.

This is thinking that obviously has nothing to do with numbers.

But it is clearly *about* something: my keys, pocket, refrigerator.

Can we understand it as a form of computation?

Gottfried Leibniz (1646-1716)

Co-inventor of the calculus (with Newton)

The first person to seriously consider the idea that ordinary thinking was computation

- the rules of *arithmetic* allow us to deal with *abstract numbers* in terms of concrete symbols

  manipulations on numeric symbols mirror relationships among the numbers being represented

- the rules of *logic* allow us to deal with *abstract ideas* in terms of concrete symbols

  manipulations on propositional symbols mirror relationships among ideas being represented

Propositions vs. sentences

Definition: A proposition is the *idea* expressed by a declarative sentence.

for example, the idea that

- my keys are in my coat pocket
- dinosaurs were warm-blooded
- somebody will fail this course

They are abstract entities (like numbers) but have special properties:

- They are considered to *hold* or *not hold*. A sentence is considered to be *true* if the proposition it expresses holds.
- They are *related to people* in various ways: people can believe them, disbelieve them, fear them, worry about them, regret them, *etc.*
- They are *related to each other* in various ways: entail, provide evidence, contradict, *etc.*

A first clue

We do not necessarily need to know what the terms in a sentence *mean* to be able to think about it.

Example: The snark was a boojum.

And now answer the following:

- What kind of thing was the snark?
- Is there anything that was a boojum?
- Was the snark either a beejum or a boojum?
- Given that no boojum is every a beejum, was the snark a beejum?

We can still figure out appropriate answers using simple rules.

Some related thoughts

My keys are in my coat pocket or on the fridge.

Nothing is in my coat pocket.

So my keys are on the fridge.

Compare to this:

Henry is in the basement or in the garden.

Nobody is in the basement.

So Henry is in the garden.

And compare to this:

Jill is married to George or Jack.

Nobody is married to George.

So Jill is married to Jack.

It’s all in the form

All three examples are really the same:

Blue thing is green thing or yellow thing.

Nothing is green thing.

So blue thing is yellow thing.

It does not matter whether

- blue thing is “my keys” or “Henry” or “Jill”
- green thing is “in my coat pocket or “in the basement” or “married to George”

Note: The thinking is the same!

The only thing that matters is that it is the same term (the blue thing) that is used in the first and third sentences, etc.

Entailment

A collection of sentences, $S_1, S_2, \ldots S_n$ *entail* a sentence S if the truth of S is implicit in the truth of the S_i.

> No matter what the terms in the S_i really mean, if the S_i sentences are all *true*, then S must also be *true*.

For example,

The snark was a boojum.

Entails: Something was a boojum.

My keys are in my coat pocket or on the fridge.

and Nothing is in my coat pocket.

Entails: My keys are on the fridge.

Can this be how we think??

Suppose we are told at a party: George is a bachelor.

Here is what is entailed:

- Somebody is a bachelor.
- George is either a bachelor or a farmer.
- Not everyone is not a bachelor.
- It is not the case that George is not a bachelor. . . .

All true, but very very *boring*!

But when *we* think about it, we think about

- George (who we may know a lot about)
- being a bachelor (which we may know a lot about)

So *our* thinking appears to depend on what the terms mean!

Using what you know

We must consider the entailments of not only George is a bachelor,
but this + a large collection of other sentences we might already know:

George was born in Boston, collects stamps.

A son of someone is a child who is male.

George is the only son of Mary and Fred.

A man is an adult male person.

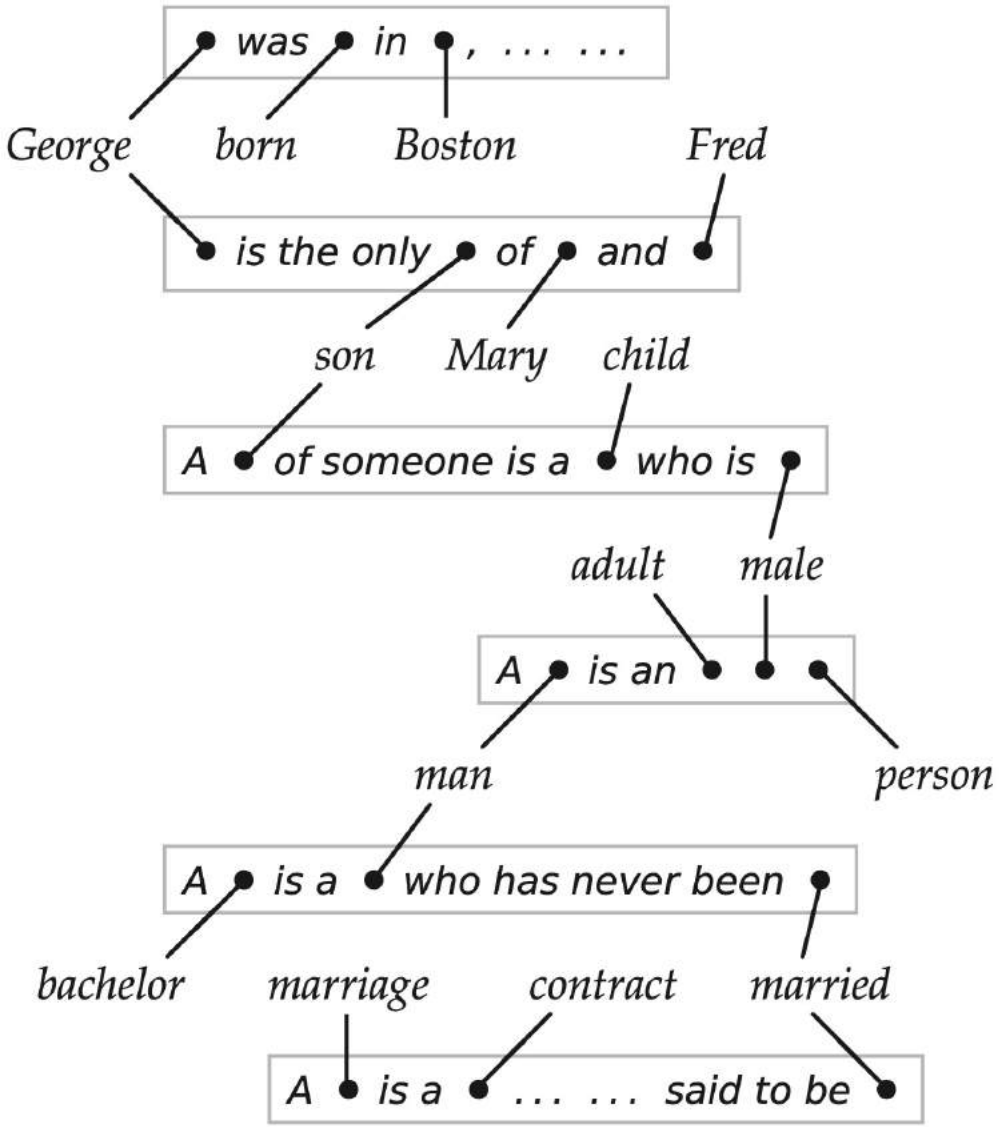
A bachelor is a man who has never been married.

A (traditional) marriage is a contract between a man and a woman, enacted by a wedding and dissolved by a divorce. While the contract is in effect, the man (called the husband) and the woman (called the wife) are said to be married.

A wedding is a ceremony where . . . bride . . . groom . . . bouquet . . .

The terms like “George” and “bachelor” and “person” and “stamps” appear in many places and link these sentences together.

The web of belief



Each sentence we believe is linked to many others by virtue of the terms that appear in them.

It is the job of logic to crawl over this web looking for connections among the terms.

Using what you know

When we include all these other facts, here are some entailments we get:

- George has never been the groom at a wedding.
- Mary has an unmarried son born in Boston.
- No woman is the wife of any of Fred's children.

→ Example of inductive bias

The conclusions are much more like those made by ordinary people.

We find where the same term appears (like we did with the blue thing, the green thing etc.).

We then apply simple rules of logic to draw conclusions.

We still do not need to know what "George" or "bachelor" means!

Now, the big step:

Imagine drawing conclusions from *millions* of such facts.

Two hypotheses

1. Much of the richness of meaning that *we* experience during thinking may be accounted for in terms of simple symbolic manipulations over a *rich* collection of represented propositions

2. To build computers systems that are versatile, flexible, modifiable, explainable, . . . it is a good idea to
   - represent much of what the system needs to know as symbolic sentences
   - perform manipulations over these sentences using the rules of symbolic logic to derive new conclusions
   - have the system act based on the conclusions it is able to derive

   Systems built this way are called *knowledge-based systems* and the collection of sentences is called its *knowledge base* (*KB*).

Summary

Thinking, as far as we are concerned, means bringing what you know to bear on what you are doing.

But how does this work? How do concrete, physical entities like *people* engage with something formless and abstract like *knowledge*?

The answer (via Leibniz) is that we engage with *symbolic representations* of that knowledge.

> We represent knowledge symbolically as a collection of sentences in a knowledge base, and then compute entailments of those sentences, as we need them.

So *computation over a knowledge base* is the direction that we will be pursuing in this course, although we will only ever deal with tiny knowledge bases here.

CPS721, Sections 051 to 091
Lecture 2

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- Back-chaining Algorithm

Atomic and Conditional Sentence Knowledge Bases

A special but very common case

Knowledge bases consisting of two sorts of sentences:

- atomic sentences: simple primitives (e.g., subject-verb-object)
- conditional sentences: of the form

  `If` P_1 `and` ... `and` P_n `then` Q

  where the P_i and Q are atomic sentences. In both cases, the sentences may contain variables (written in caps) as well as constants (lower case).

Example (abbreviation `ev` means electricVehicle)
Constant `tesla123` is a name of a specific vehicle manufactured by Tesla.
Constant `rav456` is a name of a RAV4 vehicle manufactured by Toyota.

```
tesla123 is a sedan.
rav456 is a suv.
tesla123 is electric.
If X is car and X is electric then X is ev.
If X is a suv then X is a car.
If X is a sedan then X is a car.
```

Back-chaining

Can use a simple procedure to derive new conclusions

Procedure to establish an atomic sentence Q

Either (a) locate Q itself

or (b) find a conditional sentence of the form

(`If` P_1 `and` ... `and` P_n `then` Q)

and then establish each of the P_i

End

How to deal with variables:

May want to establish (`nissanLeaf789 is ev.`)

- Find a conditional (`If --- then` Z `is ev.`)
- Can use this conditional, provided that we replace Z by `nissanLeaf789` in the P_i

Tracing some examples

Knowledge Base

```
tesla123 is a sedan.
rav456 is a suv.
tesla123 is electric.
If X is car and X is electric then X is ev.
If X is a suv then X is a car.
If X is a sedan then X is a car.
```

1. establish (`tesla123 is a sedan`)
   part (a):succeeds
   succeeds

2. establish (`rav456 is a car`)
   part (a): fails
   part (b): have (`If -- then X is a car`)
   establish (`rav456 is a suv`)
   part (a):succeeds
   succeeds
   succeeds

3. establish (`rav456 is japanese`)
   part (a): fails
   part (b): fails
   fails

Note: the conclusion is probably true. It is just not entailed by the KB

A longer trace

Knowledge Base

```
tesla123 is a sedan.
rav456 is a suv.
tesla123 is electric.
If X is car and X is electric then X is ev.
If X is a suv then X is a car.
If X is a sedan then X is a car.
```

4. establish (tesla123 is ev)
   part (a): fails
   part (b): have (If -- then X is ev)
   establish (tesla123 is a car)
   part (a): fails
   part (b): have (If X is a suv then X is a car)
   establish (tesla123 is a suv)
   part (a): fails
   part (b): fails
   fails
   /* Backtrack: use 2nd rule */
   part (b): have (If X is a sedan then X is a car)
   establish (tesla123 is a sedan)
   part (a): succeeds
   succeeds
   succeeds
   establish (tesla123 is electric)
   part (a): succeeds
   succeeds
   succeeds

Retrieval

Can also use back-chaining to locate individuals that satisfy a given property

- establish (`tesla123 is a car`) means confirm that tesla123 is a car
- establish (*Z* `is a car`) means locate an individual *Z* that is a car.

establish (*Z* `is a car`)
part (a): fails
part (b): have (`If --  then` *X* `is a car`)
establish (*Z* `is a suv`)
part (a): succeeds for *Z*=`rav456`
succeeds for *Z*=`rav456`
succeeds for *Z*=`rav456`
/* and would also for `tesla123` if we would try to retrieve another answer */

In the case where there are individuals that satisfy the current goal, we select one and continue.

But we may have to reconsider the choice later in the procedure: this is called back-tracking

A back-tracking trace

Knowledge Base

```
tesla123 is a sedan.
rav456 is a suv.
tesla123 is electric.
If X is car and X is electric then X is ev.
If X is a suv then X is a car.
If X is a sedan then X is a car.
```

5. establish (*Z* is ev)
   part (a): fails
   part (b): have (If -- then *X* is ev)
   establish (*Z* is a car)
   part (a): fails
   part (b): have (If *X* is a suv then *X* is a car)
   establish (*Z* is a suv)
   part (a): success *Z*=rav456 /*Anybody else? */
   success for *Z*=rav456
   success for *Z*=rav456

   establish (rav456 is electric)
   part (a): fails
   part (b): fails
   fails

Back-track and try to retrieve something else instead of rav456

Perhaps the goal failed because of the choice of *Z* as rav456.
Need to reconsider . . .

The trace continued ...

Knowledge Base

```
tesla123 is a sedan.
rav456 is a suv.
tesla123 is electric.
If X is car and X is electric then X is ev.
If X is a suv then X is a car.
If X is a sedan then X is a car.
```

5. establish (Z is a ev)
   part (a): fails
   part (b): have (If -- then X is a ev)
   establish (Z is a car)
   part (a): fails
   part (b): have (If X is a suv then X is a car)
   establish (Z is a suv)
   part (a): *fails* ↩ /*Backtrack: No other value for Z */
   part (b): fails
   fail

   2nd: have (If X is a sedan then X is a car)
   establish (Z is a sedan)
   part (a): succeeds for Z=tesla123
   succeeds for Z=tesla123
   succeeds (Z=tesla123)

   establish (tesla123 is electric)
   part (a): succceeds
   succeeds
   succeeds for Z=tesla123

Variable collision

One final complication due to variables:

We may have a conditional:
(If *X* is a parent of *Y* and *X* is female then *X* is a mother)
and facts about who is a parent of whom.

If we want to establish (sue is a mother),
then things will work fine (using retrieval):
establish (sue is a parent of *Y*), (sue is female)

If we want to establish (*Z* is a mother),
then things will work (maybe with backtracking):
establish (*Z* is a parent of *Y*), (*Z* is female)

But if we want to establish (*Y* is a mother),
then things do not work:
establish (*Y* is a parent of *Y*), (*Y* is female)

The solution: before using a conditional, we rename its variables so that they are always different from those in the query.

Some properties

Back-chaining has some desirable features:

1) *it is sound*, i.e., anything that can be established is logically entailed

2) *it is complete for simple sentences*,
i.e., all atomic entailments can be established assuming you eventually get to try all applicable alternatives that can be found. The latter means no cyclic rules such as

`If` *X* `is a girl` `then` *X* `is a girl`

3) *it is goal-directed*,
i.e., work your way back from what you are trying to establish towards what you know
Contrast with the undirected "forward-chaining" procedure in the "keys and coat" example

4) *it forms the basis of declarative programming in PROLOG*.
In PROLOG:

knowledge base → program
back-chaining (with back-tracking) → program execution.

CPS721, Sections 051 to 091
Lecture 3

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- Introduction to Prolog

What is Prolog?

- Prolog = PROgramming in LOGic, i.e., declarative, non-imperative programming. Radically different concept of programming from “conventional” computer languages:

  Computation = Logical deduction.

- Invented in 1971 by Frenchmen, Alain Colmerauer and Philippe Roussel, while visiting at the University of Montreal. Based on the previous research in automated theorem-proving of Robert Kowalski (IJCAI Award for Research Excellence, 2011). Purpose: natural language processing.

- Simultaneously invented by an American, Carl Hewitt, at MIT in Boston (under the name “Planner” there).

- Now a commercial product, with many available systems. We use ECLiPSe Constraint Logic Programming System owned by Cisco Technology.

- Widely used for AI applications that require symbol processing, as well as many other uses (databases, compilers).

Prolog Program

Prolog programs are simply *knowledge bases* consisting of atomic and conditional sentences as before, but with a sightly different notation.

Here is the "family" example as a Prolog program:

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

john is a child of sue.
john is a child of sam.
jane is a child of sue.
jane is a child of sam.
sue is a child of george.
sue is a child of gina.

john is male. jane is female. june is female.
sam is male. sue is female. george is male.

If X is a child of Y then Y is a parent of X.
If X is a child of Y and Y is male then Y is a father of X.
If X is male and Y is female then X is of opposite sex from Y.
If X is male and Y is female then Y is of opposite sex from X.
If X is a father of Y and Y is a parent of Z then X is a grandfather of Z.

Prolog Program: Constants and variables

A Prolog *constant* must start with a *lower case* letter, and can then be followed by any number of letters, underscores, or digits.

A constant may also be a *quoted-string*: any string of characters (except a single quote) enclosed within single quotes.

So the following are all legal constants:

```
sue opp_sex mamboNumber5 'Who are you?'
```

A Prolog *variable* must start with an *upper case* letter, and can then be followed by any number of letters, underscores, or digits.

So the following are all legal variables:

```
X P1 MyDog The_biggest_number Variable_27b
```

Prolog also has *numeric* terms, which we will return to later.

Prolog Program: Atomic Sentences (a.k.a. atoms)

Prolog programs start with basic facts:

- Marc is a baby
- Michelle is a toddler
- Marc likes ice cream

In Prolog, these would written as follows

- baby(marc).
- toddler(michelle).
- likes(marc,ice_cream).

These are called atomic sentences (or just atoms). The words `baby`, `toddler`, and `likes` are examples of Prolog predicates; `marc`, `michelle`, and `ice_cream` are examples of Prolog constants.

- The names of predicates and constants are up to you, the programmer.
- They must **not** begin with an upper-case letter or an underscore.
- Have the form `predicate(argument`$_1$`, … ,argument`$_n$`)`

A Simple Prolog Program

The simplest possible Prolog program is a collection of atomic sentences, each ending with a period.

Usually (but not necessarily), these are placed one per line to visualize that back-chaining can process them one after another.

So this is a program:

```
baby(marc).
baby(mary).
toddler(michelle).
toddler(steve).
toddler(bob).
likes(marc,michelle).
likes(bob,ray).
likes(bob,michelle).
enrolled_in(susan,cps721,2020, artificial_intelligence).
```

Prolog Program: Clauses

Simple Prolog databases are not very interesting. No intelligence!
We want programs that *reason*.

Recall the introductory material on back-chaining and conditional sentences:

- If X is male and X is a child then X is a boy.
- If X is a toddler then X is a child.

Prolog programs may (and usually will) contain such conditional sentences, which in Prolog are called general rules or clauses.

```
child(X) :- toddler(X).
```

"*X* is a child if *X* is a toddler."

```
likes(bob,X) :- child(X).
```

"Bob likes children."

```
likes(X,Y) :- baby(Y).
```

"Everyone likes babies."

```
boy(X) :- male(X), child(X).
```

"*X* is a boy if *X* is a male and *X* is a child."

Prolog Program: Clauses

The conditional sentences of Prolog have the following form:

$$head \ :\!- \ body_1, \ldots, body_n$$

where the head and each element of the body is an atom.

Note the punctuation:

- immediately after the head, there must be a *colon* then *hyphen*, `:-`.
- between each element of the body, there must be a *comma*.

If $n = 0$, the `:-` should be omitted.

In other words, an atomic sentence is just a conditional sentence where the body is empty!

How to use Prolog programs?

Prolog programs only do something in response to *queries*.

Here is the normal way of running a Prolog program:

1. We begin by preparing a file containing the Prolog program.
2. We run the Prolog system and ask it to load the Prolog program.
3. Then we repeatedly do the following:
   - we pose a query to the system
   - we wait for Prolog to return an answer

   Often we save these queries and answers in a separate file.
4. Finally we exit the Prolog system.

The details of how these steps are done vary from system to system.

Simple queries

In its simplest form, a *query* is just an atom, with or without variables, and terminated with a period.

Posing a query is asking Prolog to *establish* the atom using back-chaining, just as we did by hand before.

If the query has no variables, there are three possible outcomes:

1. Prolog answers `Yes` (or `true`): the atom can be established;

   See the query `father(sam,jane)` on the previous slide.

2. Prolog answers `No` (or `false`): the atom cannot be established, and Prolog runs out of alternatives to try;

3. Prolog does not answer: the atom cannot be established, and Prolog continues to try alternatives.

Querying a program

Examples of atomic queries and the answers Prolog returns:

?- baby(marc).

?- baby(michelle).

?- likes(bob,ray).

?- likes(ray,bob)

```
baby(marc).
baby(mary).
toddler(michelle).
toddler(steve).
toddler(bob).
likes(marc,michelle).
likes(bob,ray).
likes(bob,michelle).
enrolled_in(susan,cps721,2020, artificial_intelligence).
```

Queries with variables

If a query has variables, then there are three possible outcomes:

1. Prolog answers `No`: the atom cannot be established for any value of the variables, and Prolog runs out of alternatives to try;
2. Prolog does not answer: the atom cannot be established for any value of the variables, and Prolog continues to try alternatives;
3. Prolog displays values for the variables for which it can establish the query. At this point the user has some choices:
   - type a *space* or *return*, in which case, Prolog answers `Yes` and the query is done;
   - type a *semi-colon*, in which case, Prolog tries to find new values for the variables, with the same three possible outcomes.

Example queries with variables

```
?- father(sam,X).
X = john
Yes
```

Who is Sam the father of?

Just the first answer, please.

```
?- father(U,V).
```

Who is a father of whom?

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

Conjunctive queries

More generally, queries can be *sequences* of atoms separated by commas and again terminated by a period.

These queries are understood *conjunctively*, just like the body of a conditional sentence. (So "," plays the role of "and".)

```
?- female(F), parent(sam,F).
F = jane ;
No
```

This asks for an F such that F is female and Sam is a parent of F.

In other words:

Q: Who is a female that Sam is a parent of?

A: Jane. And this is the only value that can be found.

Negation in queries

The special symbol \+ (meaning "not") can also be used in front of an atom in a query to flip between "Yes" and "No".

```
?- child(john,george).
No
```
Is John a child of George

```
?- \+ child(john,george).
Yes
```
Is John not a child of George

```
?- parent(X,john), female(X).
X = sue
Yes
```
Who is a parent of John and female

```
?- parent(X,john), \+ female(X).
X = sam
Yes
```
Who is a parent of John and not female

Note: `female(X)` is not the same as `\+ male(X)`.

for example, the query `female(gina)` will fail, but `\+ male(gina)` will succeed since Gina is not known to be male.

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

How does Prolog answer queries (1)?

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

Before looking at more general queries and programs, let us go through the steps Prolog follows in back-chaining in more detail.

Consider: `parent(X,john), \+ female(X).`

1. We start by replacing variables in the query by new names to make sure they do not conflict with variables in the Prolog program.

   `parent(_G312,john), \+ female(_G312).`

2. We start working on the query `parent(_G312,john)`, and find the clause that relates `parent` to `child`. This leaves us with:

   `child(john,_G312), \+ female(_G312).`

3. We start working on the query `child(john,_G312)`, and find the clause `child(john,sue).` This part is tentatively done and leaves us with:

   `\+ female(sue).`

. . . continued

How does Prolog answer queries (2)?

4. We start working on the query `\+ female(sue).`
   Since this is a negated query, we consider the unnegated version.

   We start working on the query `female(sue)`, and find the clause in the program. We return success.

   Since the unnegated query succeeds, the negated query fails.
   We go back to our most recent choice point (at step 3) and reconsider.
5. We return to the query `child(john,_G312)`, and this time, we find the clause `child(john,sam).` This leaves us with:

   `\+ female(sam).`
6. We start working on the query `\+ female(sam).`
   Since this is a negated query, we try the unnegated version.

   We start working on the query `female(sam)`, but we cannot find anything that matches. We return failure.

   Since the unnegated query fails, the negated query succeeds.
   This leaves us nothing left to do so.
7. We return *success*, noting that the value of `X` is `sam`.

```
% This is the Prolog version of the family example
child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

Caution: variables and negation

```
?- parent(X,john), \+ female(X).
```

```
?- \+ female(X), parent(X,john).
```

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

What is going here?

In the second case, Prolog starts by working on `\+ female(X)`.
Since `female(X)` succeeds, the negated version fails.
We can go no further with the query!

To get `\+ female(X)` to succeed (as in the first case), we need to already have a value for the `X` (e.g. `X=sam`).

Moral: When using a negated query with variables, the variables should already have tentative values from earlier queries.

The equality predicate

Prolog allows elements in a query of the form $term_1 = term_2$, where the terms are either constants or variables. This query succeeds when the terms are equal, and fails otherwise.

```
?- Z=Y, X=jack, Y=X, \+ jill=Z.
Z = jack
Y = jack
X = jack
Yes

?- X=Y, jack=X, \+ Y=jack.
No

?- Z=Y.
Z = _G180
Y = _G180
Yes
```

These answers mean that the values are equal but otherwise unconstrained.

Using equality

You should never have to use an *unnegated equality* predicate:

Instead of: `child(john,X), child(jane,Y), X=Y.`
use: `child(john,X), child(jane,X).`

However, *negated equality* does come in handy:

```
?- parent(sam,X), \+ X=john.
X = jane
Yes
```

Is Sam the parent of someone other than John?

```
?- male(X), male(Y), male(Z), \+ X=Y, \+ X=Z, \+ Y=Z.
X = john
Y = sam
Z = george
Yes
```

Are there at least 3 males?

This gives us a simple form of *counting*.

Review: terms

Before looking at Prolog programs in more detail, we review what we have seen so far, with a few minor extensions.

- A *constant* is one of the following:
  - a string of characters enclosed within single quotes.
  - a lower case letter optionally followed by letters, digits and underscores.
- A *variable* is an upper case letter optionally followed by letters, digits and underscores.
- A *number* is a sequence of one or more digits, optionally preceded by a minus sign, and optionally containing a decimal point.
- A *term* is a constant, variable, or number.

Review: queries and programs

- A *predicate* is written like a constant.
- An *atom* is a predicate optionally followed by terms enclosed within parentheses and separated by commas.
- A *literal* is one of the following, optionally preceded by a \+ symbol:
  - an atom;
  - two terms separated by the = symbol.
- A *query* is a sequence of one or more literals, separated by commas, and terminated by a period.
- A *clause* is one of the following:
  - an atom terminated with a period;
  - an atom followed by the :- symbol followed by a query.
- A *program* is a sequence of one or more clauses.

Back-chaining, revisited

Having looked at Prolog queries in some detail, we now examine Prolog back-chaining, starting with a very simple program.

```
%  This is a program about who likes what kinds of food.
likes(john,pizza).                    % John likes pizza.
likes(john,sushi).                    % John likes sushi.
likes(mary,sushi).                    % Mary likes sushi.
likes(paul,X) :- likes(john,X).       % Paul likes what John likes.
likes(X,icecream).                    % Everybody likes ice cream.
```

We start by looking at how we decide which clauses to use in answering queries.

The way clauses are selected during back-chaining is through a matching process called *unification*.

Unification

Two atoms with distinct variables are said to *unify* if there is a substitution for the variables that makes the atoms identical.

- a query such as `likes(john,Y)` unifies with `likes(john,pizza)` in the first clause of the program, for `Y=pizza`.
- a query such as `likes(paul,pizza)` unifies with `likes(paul,X)` in the fourth clause, for `X=pizza`.
- a query such as `likes(jane,Y)` unifies with `likes(X,icecream)` in the last clause, for `X=jane` and `Y=icecream`.

In all cases, the queries will eventually succeed!

Note that both the query and the head of a clause from the program may contain variables.

Examples of unification

The following pairs of atoms unify:

- `p(b,X,b)` and `p(Y,a,b)` for `X=a` and `Y=b`
- `p(X,b,X)` and `p(a,b,Y)` for `X=a` and `Y=a`
- `p(b,X,b)` and `p(Y,Z,b)` for `X=Z` and `Y=b`
- `p(X,Z,X,Z)` and `p(Y,W,a,Y)` for `X=a`, `Z=a`, `Y=a`, and `W=a`.

The following pairs of atoms do not unify:

- `p(b,X,b)` and `p(b,Y)`
- `p(b,X,b)` and `p(Y,a,a)`
- `p(X,b,X)` and `p(a,b,b)`
- `p(X,b,X,a)` and `p(Y,Z,Z,Y)`

Renaming variables

```
%  This is a program about who likes what kinds of food.
likes(john,pizza).                    % John likes pizza.
likes(john,sushi).                    % John likes sushi.
likes(mary,sushi).                    % Mary likes sushi.
likes(paul,X) :- likes(john,X).       % Paul likes what John likes.
likes(X,icecream).                    % Everybody likes ice cream.
```

Unification is not concerned with where the two atoms come from (which one is from a query, and which one is from a program).

However, during back-chaining, Prolog *renames* the variables in the clauses of the program before attempting unification.

Example: Consider the query `likes(X,pizza), \+ X=john`.

Prolog first finds `likes(john,pizza)`, but this eventually fails.

It eventually considers the clause whose head is `likes(paul,X)`, but this cannot unify with `likes(X,pizza)` as is.

So Prolog renames the variable `X` in that clause to a totally new variable, for example, `X1`.

Then `likes(X,pizza)` unifies with `likes(paul,X1)`, and the query goes on to eventually succeed.

Back-chaining: summary

Here is a slightly more accurate picture of how back-chaining works to establish a conjunctive query $A_1, A_2, \ldots, A_n$.

1. If $n = 0$, there's nothing to do, exit with success.
2. Otherwise, try each clause in the *program* from top to bottom in turn:
   (a) Assume that the current clause has head H and body $B_1, \ldots B_m$, (where for atomic sentences, the body is empty and so $m = 0$).
   (b) Rename all the variables in the clause.
   (c) Test to see if the head H *unifies* with the first atom A_1. If it does not, move on to the next clause.
   (d) Otherwise, try to establish $B_1^*, \ldots, B_m^*, A_2^*, \ldots A_n^*$, where the * means the result of replacing variables by their values from the unification.
   (e) If this works, exit with success; if not, go on to the next clause.
3. If you get this far, you've tried all the clauses in the program. Return failure.

CPS721, Sections 051 to 091
Lecture 4

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- Prolog Rules

Prolog Program

Prolog programs are simply *knowledge bases* consisting of atomic and conditional sentences as before, but with a sightly different notation.

Here is the "family" example as a Prolog program:

```
% This is the Prolog version of the family example

child(john,sue).   child(john,sam).
child(jane,sue).   child(jane,sam).
child(sue,george).   child(sue,gina).

male(john).   male(sam).     male(george).
female(sue).  female(jane).  female(june).

parent(Y,X) :- child(X,Y).
father(Y,X) :- child(X,Y), male(Y).
opp_sex(X,Y) :- male(X), female(Y).
opp_sex(Y,X) :- male(X), female(Y).
grand_father(X,Z) :- father(X,Y), parent(Y,Z).
```

john is a child of sue.
john is a child of sam.
jane is a child of sue.
jane is a child of sam.
sue is a child of george.
sue is a child of gina.

john is male. jane is female. june is female.
sam is male. sue is female. george is male.

If X is a child of Y then Y is a parent of X.
If X is a child of Y and Y is male then Y is a father of X.
If X is male and Y is female then X is of opposite sex from Y.
If X is male and Y is female then Y is of opposite sex from X.
If X is a father of Y and Y is a parent of Z then X is a grandfather of Z.

How to use Prolog programs?

Prolog programs only do something in response to *queries*.

Here is the normal way of running a Prolog program:

1. We begin by preparing a file containing the Prolog program.
2. We run the Prolog system and ask it to load the Prolog program.
3. Then we repeatedly do the following:
   - we pose a query to the system
   - we wait for Prolog to return an answer

   Often we save these queries and answers in a separate file.
4. Finally we exit the Prolog system.

The details of how these steps are done vary from system to system.

Back-chaining, revisited

Having looked at Prolog queries in some detail, we now examine Prolog back-chaining, starting with a very simple program.

```
%  This is a program about who likes what kinds of food.
likes(john,pizza).                    % John likes pizza.
likes(john,sushi).                    % John likes sushi.
likes(mary,sushi).                    % Mary likes sushi.
likes(paul,X) :- likes(john,X).       % Paul likes what John likes.
likes(X,icecream).                    % Everybody likes ice cream.
```

We start by looking at how we decide which clauses to use in answering queries.

The way clauses are selected during back-chaining is through a matching process called *unification*.

Unification

Two atoms with distinct variables are said to *unify* if there is a substitution for the variables that makes the atoms identical.

- a query such as `likes(john,Y)` unifies with `likes(john,pizza)` in the first clause of the program, for `Y=pizza`.
- a query such as `likes(paul,pizza)` unifies with `likes(paul,X)` in the fourth clause, for `X=pizza`.
- a query such as `likes(jane,Y)` unifies with `likes(X,icecream)` in the last clause, for `X=jane` and `Y=icecream`.

In all cases, the queries will eventually succeed!

Note that both the query and the head of a clause from the program may contain variables.

Renaming variables

```
%  This is a program about who likes what kinds of food.
likes(john,pizza).                    % John likes pizza.
likes(john,sushi).                    % John likes sushi.
likes(mary,sushi).                    % Mary likes sushi.
likes(paul,X) :- likes(john,X).       % Paul likes what John likes.
likes(X,icecream).                    % Everybody likes ice cream.
```

Unification is not concerned with where the two atoms come from (which one is from a query, and which one is from a program).

However, during back-chaining, Prolog *renames* the variables in the clauses of the program before attempting unification.

Example: Consider the query `likes(X,pizza), \+ X=john`.

Prolog first finds `likes(john,pizza)`, but this eventually fails.

It eventually considers the clause whose head is `likes(paul,X)`, but this cannot unify with `likes(X,pizza)` as is.

So Prolog renames the variable `X` in that clause to a totally new variable, for example, `X1`.

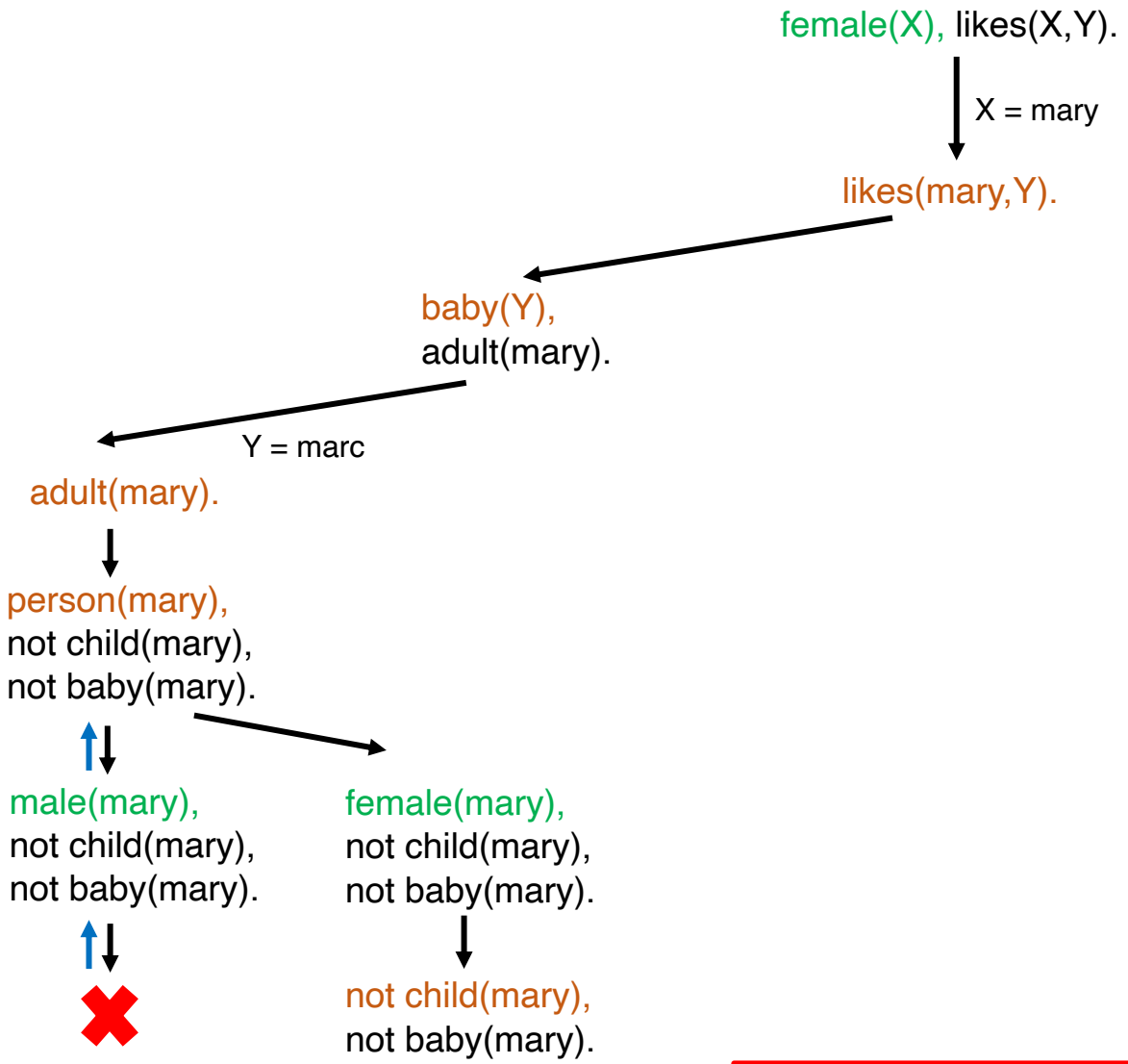
Then `likes(X,pizza)` unifies with `likes(paul,X1)`, and the query goes on to eventually succeed.

Back-chaining: summary

Here is a slightly more accurate picture of how back-chaining works to establish a conjunctive query $A_1, A_2, \ldots, A_n$.

1. If $n = 0$, there's nothing to do, exit with success.
2. Otherwise, try each clause in the *program* from top to bottom in turn:
   (a) Assume that the current clause has head H and body $B_1, \ldots B_m$, (where for atomic sentences, the body is empty and so $m = 0$).
   (b) Rename all the variables in the clause.
   (c) Test to see if the head H *unifies* with the first atom A_1. If it does not, move on to the next clause.
   (d) Otherwise, try to establish $B_1^*, \ldots, B_m^*, A_2^*, \ldots A_n^*$, where the * means the result of replacing variables by their values from the unification.
   (e) If this works, exit with success; if not, go on to the next clause.
3. If you get this far, you've tried all the clauses in the program. Return failure.

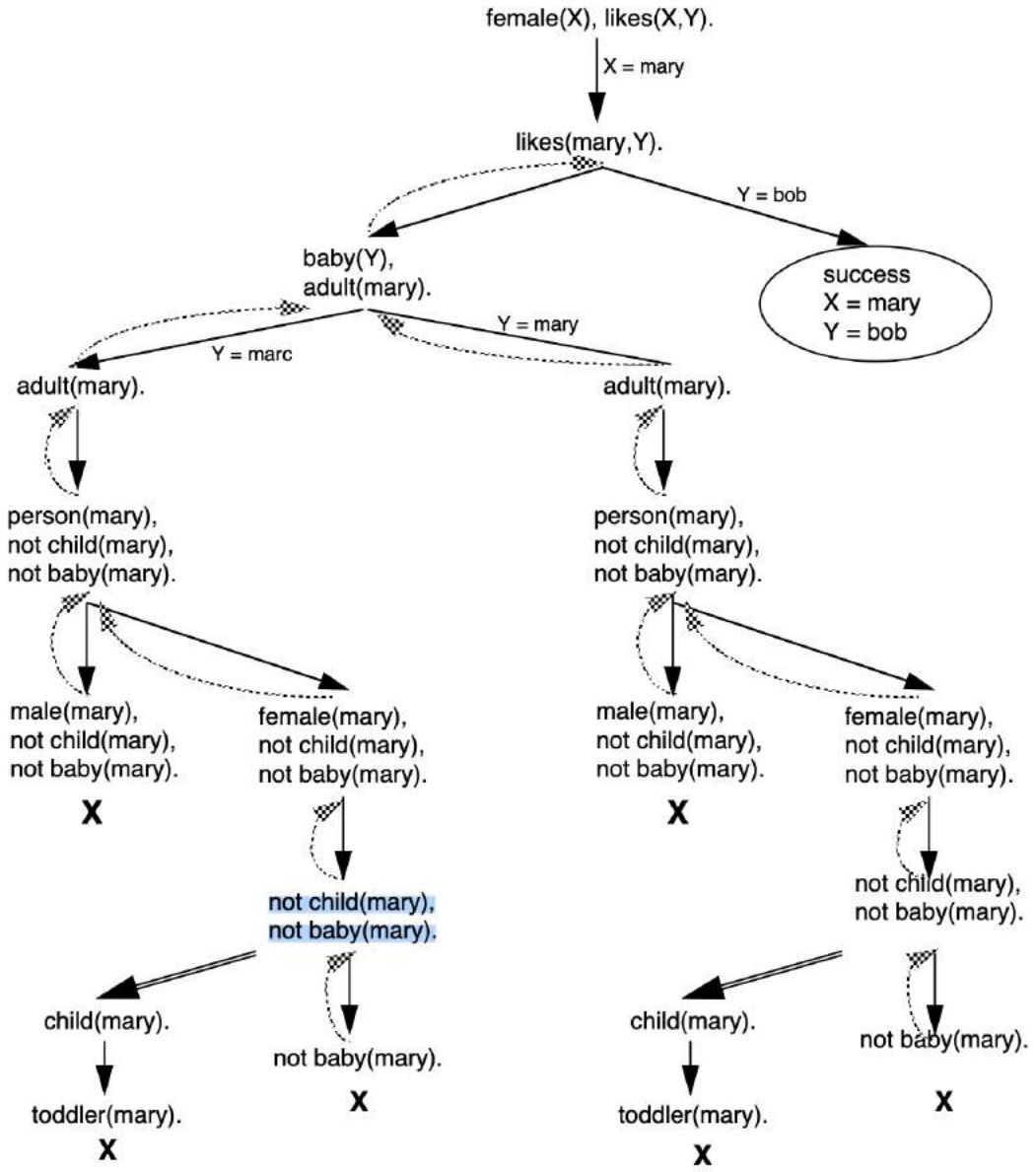
One More Example Using Evaluation Tree




```
baby(marc). baby(mary).

toddler(michelle). toddler(steve).
toddler(bob).

likes(marc,michelle). likes(bob,ray).
likes(bob,michelle).

male(ray). male(marc). male(bob).
male(steve).

female(mary). female(michelle).

child(X) :- toddler(X).

boy(X) :- male(X), child(X).
girl(X) :- female(X), child(X).

person(X) :- male(X).
person(X) :- female(X).

adult(X) :- person(X), not child(X),
                    not baby(X).

likes(X,Y) :- baby(Y), adult(X).
likes(X,bob). /* Everyone likes bob. */
```

Complete the rest of the tree as an exercise!

One More Example Using Evaluation Tree




```
baby(marc). baby(mary).

toddler(michelle). toddler(steve).
toddler(bob).

likes(marc,michelle). likes(bob,ray).
likes(bob,michelle).

male(ray). male(marc). male(bob).
male(steve).

female(mary). female(michelle).

child(X) :- toddler(X).

boy(X) :- male(X), child(X).
girl(X) :- female(X), child(X).

person(X) :- male(X).
person(X) :- female(X).

adult(X) :- person(X), not child(X),
                    not baby(X).

likes(X,Y) :- baby(Y), adult(X).
likes(X,bob). /* Everyone likes bob. */
```

Writing meaningful programs in Prolog

We can now consider writing much more complex Prolog programs.

It is important to consider carefully what we intend the predicates to *mean*, and to make sure that the clauses we write are all *true*.

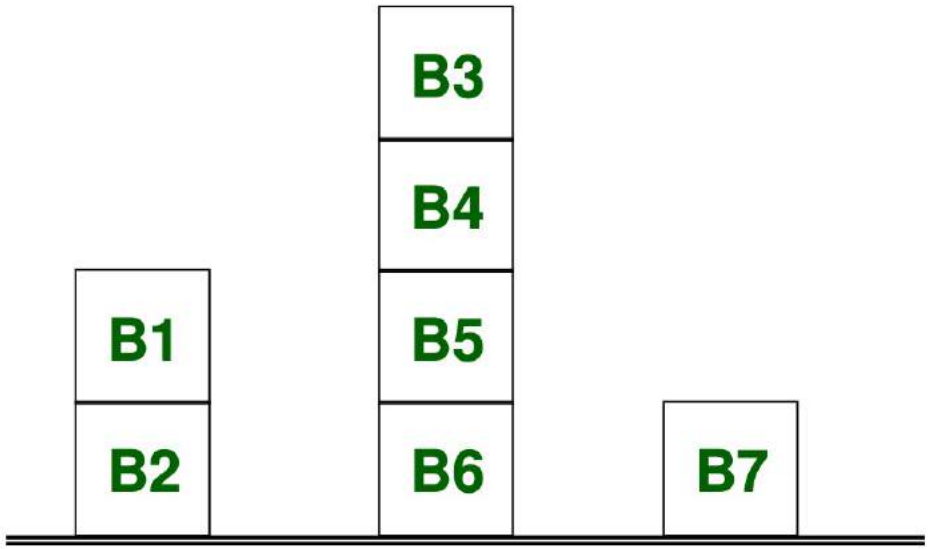
We also need to consider if we have included *enough* clauses to allow Prolog to draw appropriate conclusions involving the predicates.

Finally, we need to consider how *back-chaining* will use the clauses when actually establishing conclusions.

To summarize, we need

1. the truth, and nothing but;
2. the whole truth;
3. presented in the right form for back-chaining.

A bigger example: a world of blocks

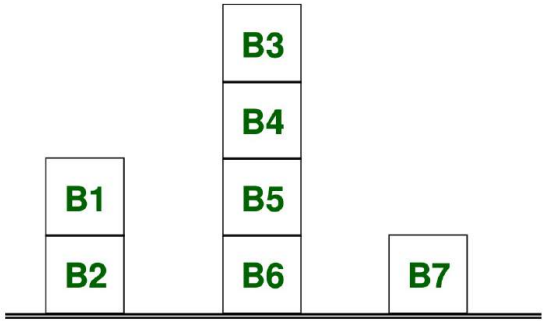


We would like to describe the scene and get Prolog to determine that

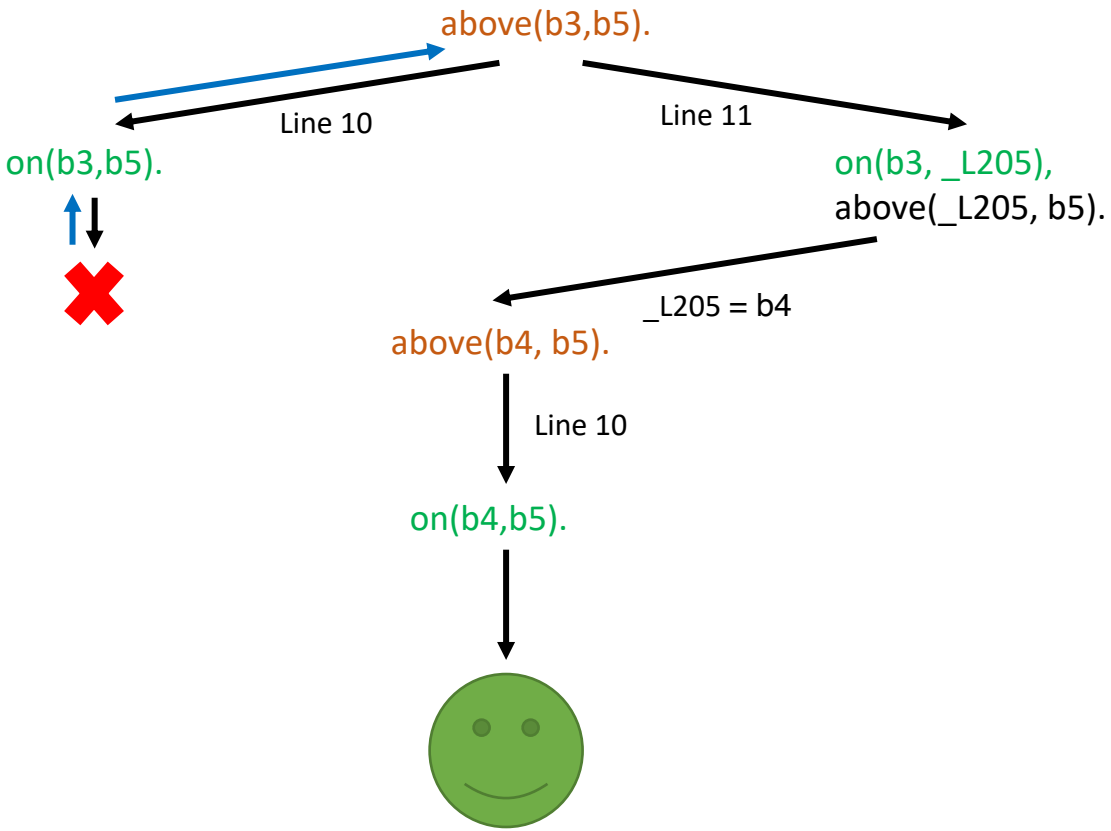
- Block 3 is above Block 5
- Block 1 is to the left of Block 7
- Block 4 is to the right of Block 2

A bigger example: a world of blocks

```
% on(X,Y) means that block X is directly on top of block Y.                1
on(b1,b2).   on(b3,b4).   on(b4,b5).   on(b5,b6).                           2

% just_left(X,Y) means that blocks X and Y are on the table                 4
% and that X is immediately to the left of Y.                               5
just_left(b2,b6).   just_left(b6,b7).                                       6

% above(X,Y) means that block X is somewhere above block Y                  8
% in the pile where Y occurs.                                               9
above(X,Y) :- on(X,Y).                                                      10
above(X,Y) :- on(X,Z), above(Z,Y).                                          11

% left(X,Y) means that block X is somewhere to the left                     13
% of block Y but perhaps higher or lower than Y.                            14
left(X,Y) :- just_left(X,Y).                                                15
left(X,Y) :- just_left(X,Z), left(Z,Y).                                     16
left(X,Y) :- on(X,Z), left(Z,Y).     % leftmost is on something.           17
left(X,Y) :- on(Y,Z), left(X,Z).     % rightmost is on something.          18

% right(X,Y) is the opposite of left(X,Y).                                  20
right(Y,X) :- left(X,Y).                                                    21
```



Back-chaining and recursion




```
% on(X,Y) means that block X is directly on top of block Y.
on(b1,b2).   on(b3,b4).   on(b4,b5).   on(b5,b6).

% just_left(X,Y) means that blocks X and Y are on the table
% and that X is immediately to the left of Y.
just_left(b2,b6).   just_left(b6,b7).

% above(X,Y) means that block X is somewhere above block Y
% in the pile where Y occurs.
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).

% left(X,Y) means that block X is somewhere to the left
% of block Y but perhaps higher or lower than Y.
left(X,Y) :- just_left(X,Y).
left(X,Y) :- just_left(X,Z), left(Z,Y).
left(X,Y) :- on(X,Z), left(Z,Y).     % leftmost is on something.
left(X,Y) :- on(Y,Z), left(X,Z).     % rightmost is on something.

% right(X,Y) is the opposite of left(X,Y).
right(Y,X) :- left(X,Y).
```

Recursion

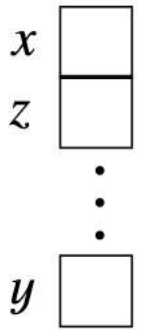
The example on the previous slide makes use of *recursion*.

A recursive clause is one in which the predicate of the head is the same as a predicate mentioned in the body.

Line 11 in the program:

```
above(X,Y) :- on(X,Z), above(Z,Y).
```

If x is on z and z is above y, then x is above y.



Most modern programming languages (Java, Python, etc.) provide recursion, which is usually considered to be an advanced technique.

In fact, it is really quite simple and lies at the heart of Prolog programming.

Writing recursive programs

You can recognize that recursion is needed when you have a predicate that involves using another predicate *some number* of times.

For example, for a block x to be above a block y, there must be some number k of intermediate blocks such that the initial x is on x_1 which is on x_2 *etc.*, and where the final x_k is on y. (The k can be 0.)

When writing the clauses for a recursive predicate, we need to somehow take care of *all* the possible cases for k.

in our scene, k might be 0 or 3 or 7 or 20 or 155,262

Q: How can we do this, since are *infinitely many* possibilities?

A: We use a form of *mathematical induction*!

Non-terminating programs

Using recursion, it is possible to write programs that go on *forever*.

For example, instead of line 11, suppose we had written this:

```
above(X,Y) :- above(Z,Y), on(X,Z).
```

What this says is *true*: if z is above y and x is on z, then x is above y.

However, if we have the query `above(b2,b1)`, we get the following:

```
above(b2,b1)    ⇒
above(Z0,b1), on(b2,Z0)    ⇒
above(Z1,b1), on(Z0,Z1), on(b2,Z0)    ⇒
above(Z2,b1), on(Z1,Z2), on(Z0,Z1), on(b2,Z0)    ⇒    ...
```

Eventually Prolog reports that it has run out of memory!

Variable renaming when tracing by writing

When you (as opposed to Prolog) are using a program clause and a subgoal to establish a new subgoal, make sure the clause's variables are different from the subgoal's variables.

In fact, Prolog does this automatically by storing clauses using internal variables which it guarantees to be different from any other variable in any other program clause or query.

Therefore, you have to rename variables in Prolog rules, only when you trace by hand how the back-chaing algorithm evaluates a query w.r.t. a recursive program.

Exercise: do it at home!
Use back-chaining to evaluate the query **?- right(b7,b2)**.
Draw a back-chaining tree as explained in class, show values retrieved for variables on every step, show back-tracking explicitly in all cases. Verify if your work is correct w.r.t. Slide 13 posted on D2L.

Avoiding non-termination

When writing recursive programs, there is no simple way to *guarantee* that they will terminate.

However, a good rule of thumb is that when a clause is recursive, the body should contain at least one atom *before* the recursive one to provide a new value for one of the variables.

Instead of `above(X,Y) :- above(Z,Y), on(X,Z).`

we should write `above(X,Y) :- on(X,Z), above(Z,Y).`

⇑

These two mean the same thing in English, and together with line 10, they correctly characterize the predicate above.

But because of the left-to-right order of Prolog, the second one will try the recursive predicate only after it has found a value for the variable Z.

Algorithm efficiency

There is no easy way to write programs and be sure that they are not doing excessive work.

A good part of Computer Science involves analyzing computational problems and coming up with *effective* ways of solving them using computers.

= comparing different *algorithms* for solving a problem.

Each problem has to be approached on its own terms.

In Prolog, we would need to think about what sorts of approaches will allow back-chaining to do its job effectively.

In this course, we will *not* spend a lot of time worrying about algorithms!

CPS721, Sections 051 to 091
Lecture 5

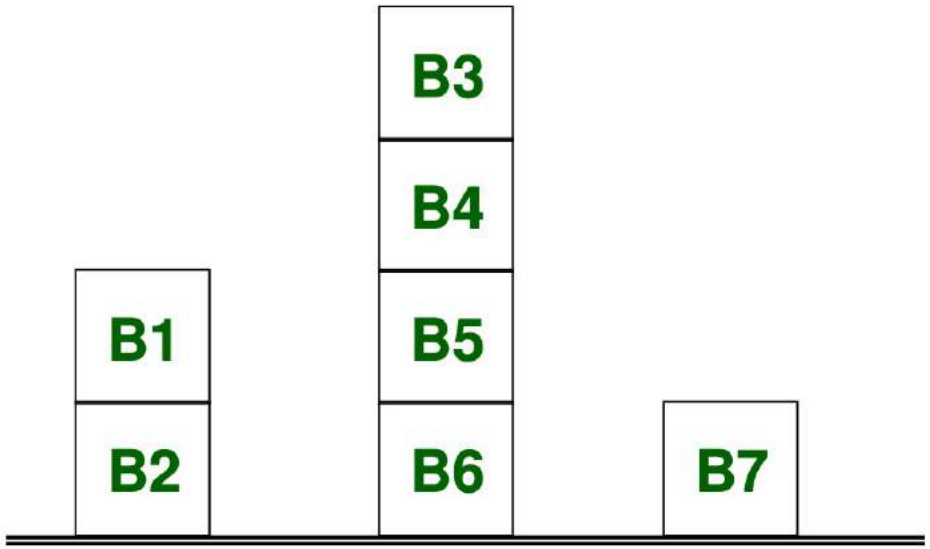
Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Review Recursion**
- More on Prolog with Examples

A bigger example: a world of blocks

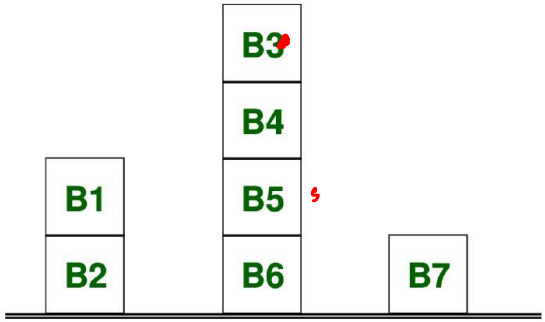


We would like to describe the scene and get Prolog to determine that

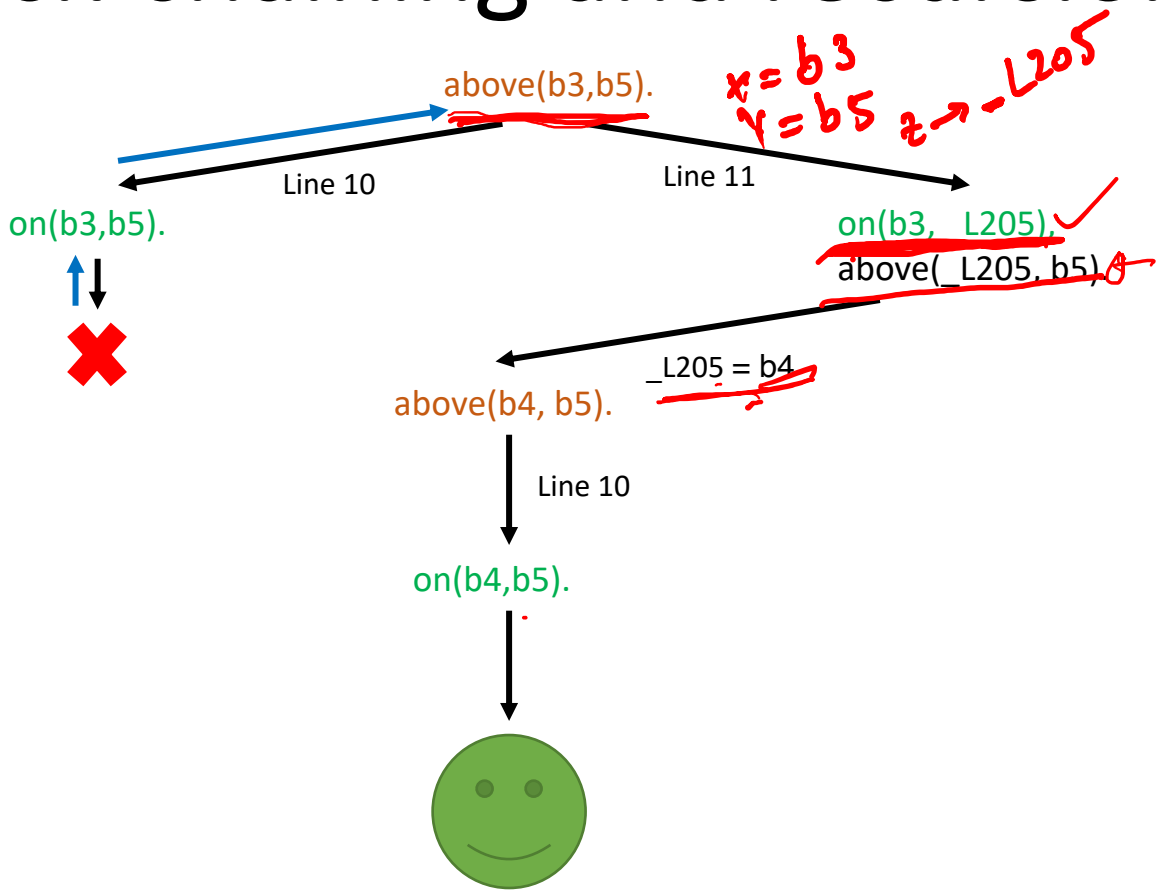
- Block 3 is above Block 5
- Block 1 is to the left of Block 7
- Block 4 is to the right of Block 2

A bigger example: a world of blocks

```
% on(X,Y) means that block X is directly on top of block Y.                1
on(b1,b2).   on(b3,b4).   on(b4,b5).   on(b5,b6).                           2

% just_left(X,Y) means that blocks X and Y are on the table                 4
% and that X is immediately to the left of Y.                               5
just_left(b2,b6).   just_left(b6,b7).                                       6

% above(X,Y) means that block X is somewhere above block Y                  8
% in the pile where Y occurs.                                               9
above(X,Y) :- on(X,Y).                                                      10
above(X,Y) :- on(X,Z), above(Z,Y).                                          11

% left(X,Y) means that block X is somewhere to the left                     13
% of block Y but perhaps higher or lower than Y.                            14
left(X,Y) :- just_left(X,Y).                                                15
left(X,Y) :- just_left(X,Z), left(Z,Y).                                     16
left(X,Y) :- on(X,Z), left(Z,Y).     % leftmost is on something.           17
left(X,Y) :- on(Y,Z), left(X,Z).     % rightmost is on something.          18

% right(X,Y) is the opposite of left(X,Y).                                  20
right(Y,X) :- left(X,Y).                                                    21
```



Back-chaining and recursion




```
% on(X,Y) means that block X is directly on top of block Y.
on(b1,b2).   on(b3,b4).   on(b4,b5).   on(b5,b6).

% just_left(X,Y) means that blocks X and Y are on the table
% and that X is immediately to the left of Y.
just_left(b2,b6).   just_left(b6,b7).

% above(X,Y) means that block X is somewhere above block Y
% in the pile where Y occurs.
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).

% left(X,Y) means that block X is somewhere to the left
% of block Y but perhaps higher or lower than Y.
left(X,Y) :- just_left(X,Y).
left(X,Y) :- just_left(X,Z), left(Z,Y).
left(X,Y) :- on(X,Z), left(Z,Y).     % leftmost is on something.
left(X,Y) :- on(Y,Z), left(X,Z).     % rightmost is on something.

% right(X,Y) is the opposite of left(X,Y).
right(Y,X) :- left(X,Y).
```

Recursion

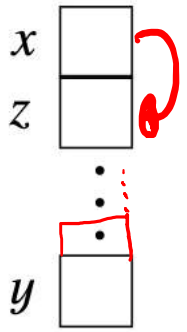
The example on the previous slide makes use of *recursion*.

A recursive clause is one in which the predicate of the head is the same as a predicate mentioned in the body.

Line 11 in the program:

```
above(X,Y) :- on(X,Z), above(Z,Y).
```

If x is on z and z is above y, then x is above y.



Most modern programming languages (Java, Python, etc.) provide recursion, which is usually considered to be an advanced technique.

In fact, it is really quite simple and lies at the heart of Prolog programming.

Writing recursive programs

You can recognize that recursion is needed when you have a predicate that involves using another predicate *some number* of times.

For example, for a block x to be above a block y, there must be some number k of intermediate blocks such that the initial x is on x_1 which is on x_2 *etc.*, and where the final x_k is on y. (The k can be 0.)

When writing the clauses for a recursive predicate, we need to somehow take care of *all* the possible cases for k.

in our scene, k might be 0 or 3 or 7 or 20 or 155,262

Q: How can we do this, since are *infinitely many* possibilities?

A: We use a form of *mathematical induction*!

Non-terminating programs

original line 11
above(X,Y) :- on(X,Z), above(Z,Y)

Using recursion, it is possible to write programs that go on *forever*.

For example, instead of line 11, suppose we had written this:

```
above(X,Y) :- above(Z,Y), on(X,Z).
```

What this says is *true*: if z is above y and x is on z, then x is above y.

However, if we have the query `above(b2,b1)`, we get the following:

```
above(b2,b1)    ⇒
above(Z0,b1), on(b2,Z0)    ⇒
above(Z1,b1), on(Z0,Z1), on(b2,Z0)    ⇒
above(Z2,b1), on(Z1,Z2), on(Z0,Z1), on(b2,Z0)    ⇒    ...
```

Eventually Prolog reports that it has run out of memory!

Variable renaming when tracing by writing

When you (as opposed to Prolog) are using a program clause and a subgoal to establish a new subgoal, make sure the clause's variables are different from the subgoal's variables.

In fact, Prolog does this automatically by storing clauses using internal variables which it guarantees to be different from any other variable in any other program clause or query.

Therefore, you have to rename variables in Prolog rules, only when you trace by hand how the back-chaing algorithm evaluates a query w.r.t. a recursive program.

Exercise: do it at home!
Use back-chaining to evaluate the query **?- right(b7,b2)**.
Draw a back-chaining tree as explained in class, show values retrieved for variables on every step, show back-tracking explicitly in all cases. Verify if your work is correct w.r.t. Slide 13 posted on D2L.

Avoiding non-termination

When writing recursive programs, there is no simple way to *guarantee* that they will terminate.

However, a good rule of thumb is that when a clause is recursive, the body should contain at least one atom *before* the recursive one to provide a new value for one of the variables.

Instead of
```
above(X,Y) :- above(Z,Y), on(X,Z).
```
we should write
```
above(X,Y) :- on(X,Z), above(Z,Y).
```

These two mean the same thing in English, and together with line 10, they correctly characterize the predicate above.

But because of the left-to-right order of Prolog, the second one will try the recursive predicate only after it has found a value for the variable Z.

Algorithm efficiency

There is no easy way to write programs and be sure that they are not doing excessive work.

A good part of Computer Science involves analyzing computational problems and coming up with *effective* ways of solving them using computers.

= comparing different *algorithms* for solving a problem.

Each problem has to be approached on its own terms.

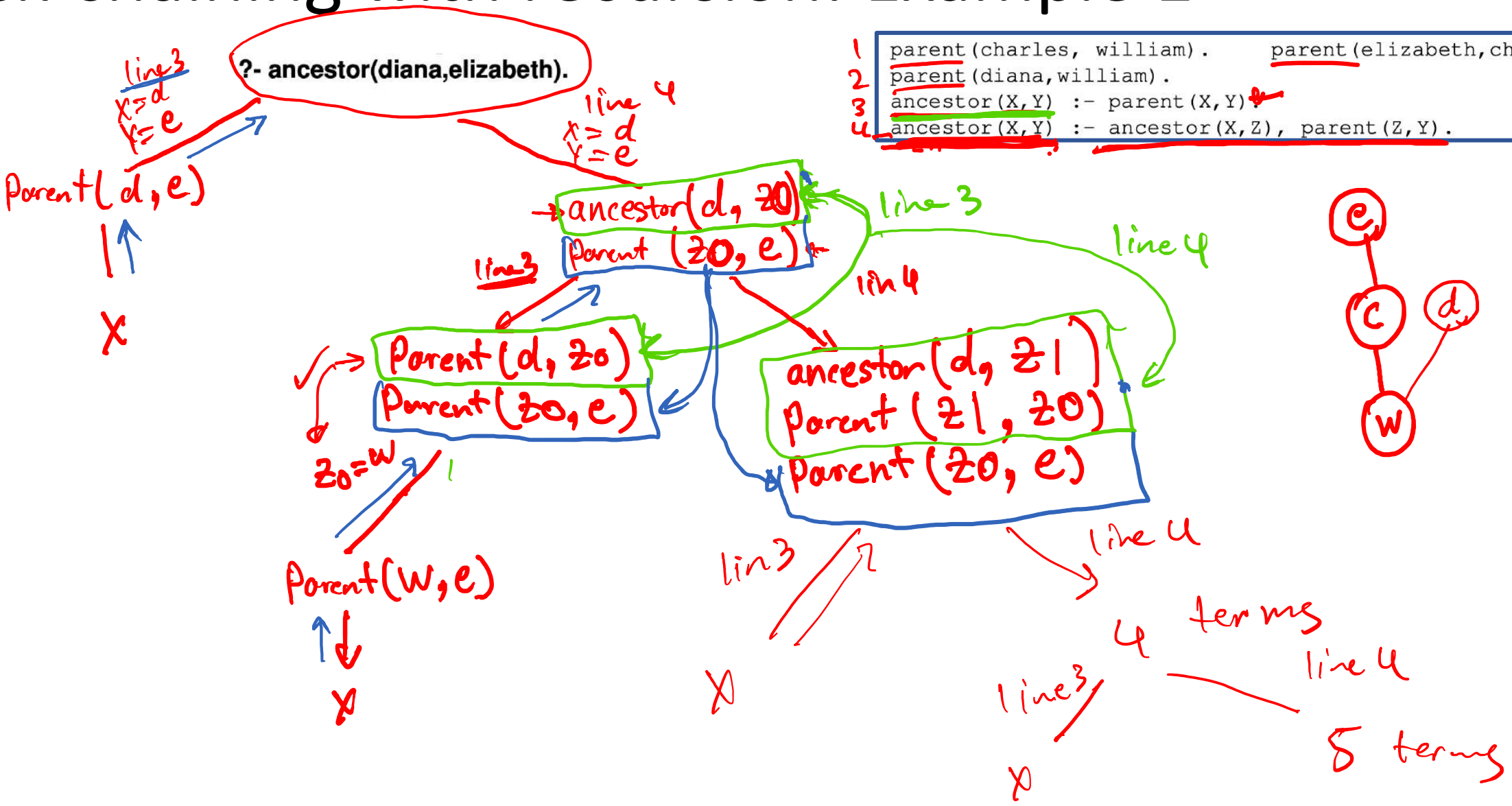
In Prolog, we would need to think about what sorts of approaches will allow back-chaining to do its job effectively.

In this course, we will *not* spend a lot of time worrying about algorithms!

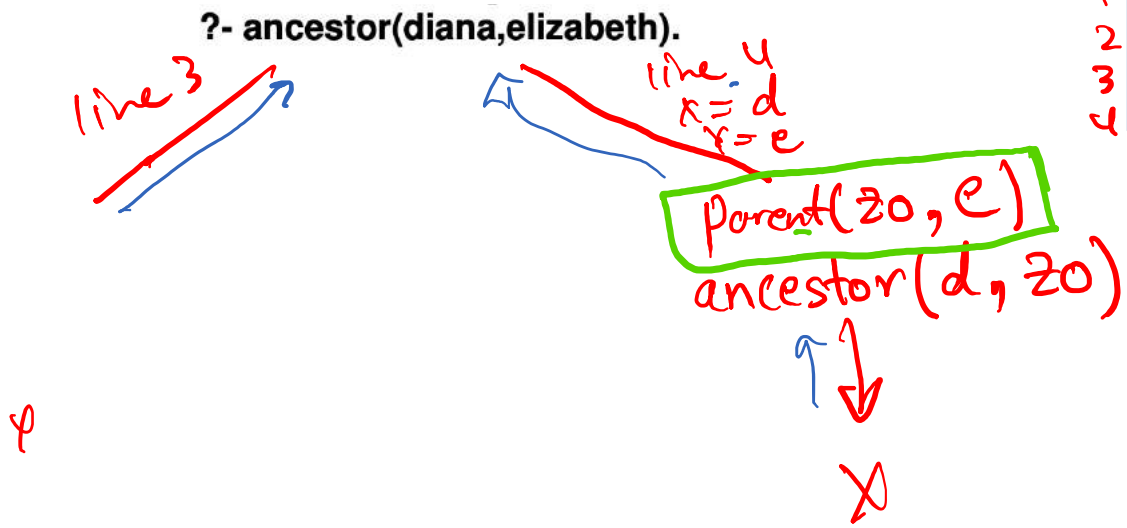
Agenda

- Review Recursion
- **More on Prolog with Examples**

Back-chaining with recursion: Example 1



How can we fix example 1?




```
1 parent(charles, william).     parent(elizabeth,charles).
2 parent(diana,william).
3 ancestor(X,Y) :- parent(X,Y).
4 ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
```

Change line 4 to

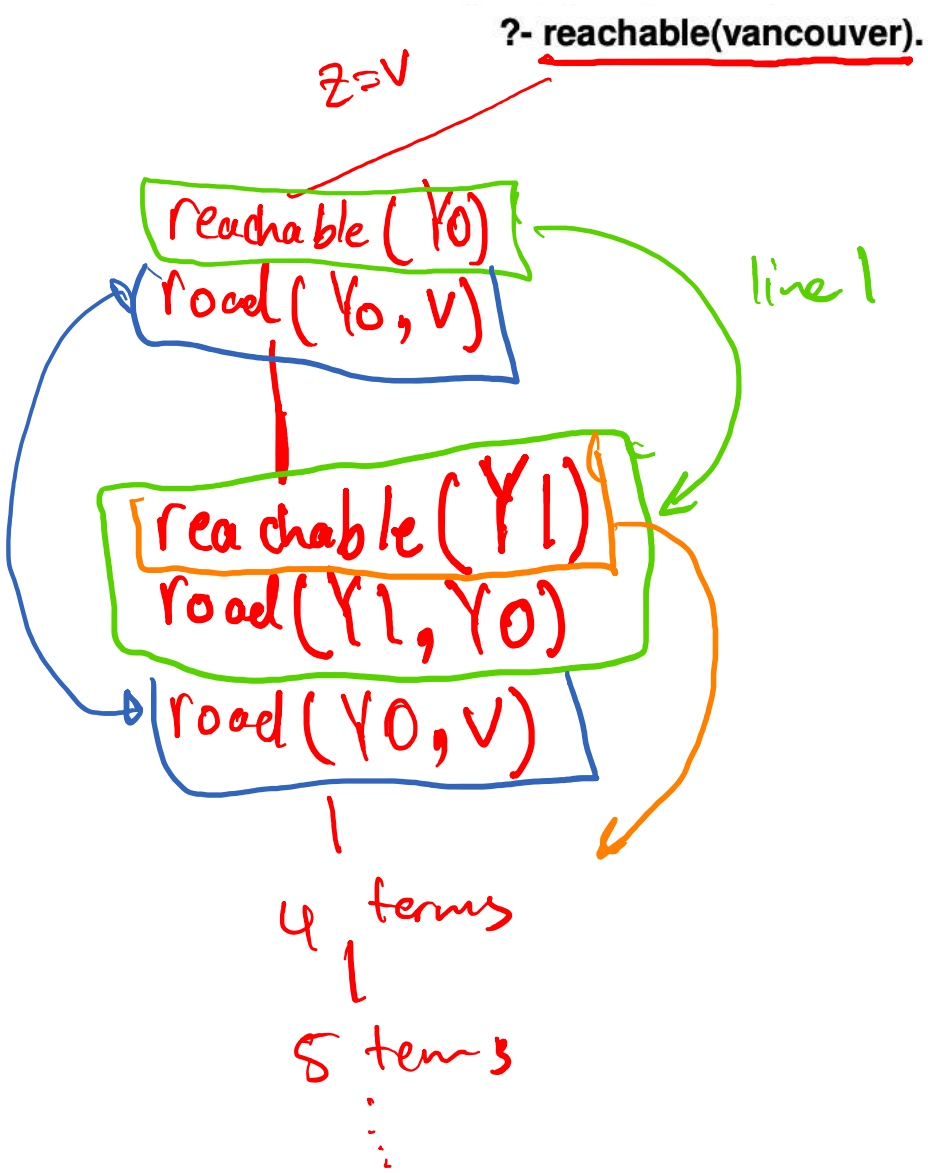
ancestor(X, Y) :- parent(Z, Y),
ancestor(X, Z)

Back-chaining with recursion: Example 2

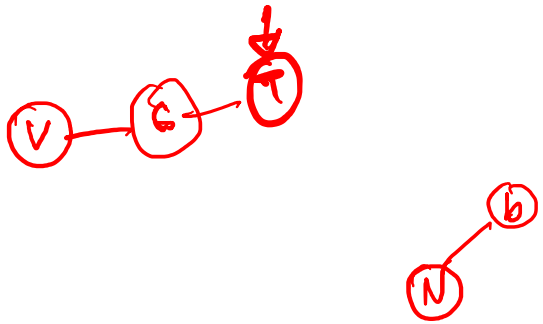
Consider three predicates: *reachable*(*C*) means a city *C* is reachable, e.g., by car from Toronto, *init_loc*(*X*) means a city *X* is an initial location, and *road*(*X*, *Y*) means there is a highway from a city *X* to a city *Y*.

```
reachable(Z) :- reachable(Y), road(Y,Z).
reachable(X) :- init_loc(X).
init_loc(toronto).
road(toronto,calgary).
road(calgary,vancouver).
road(new_york,boston).
```

Back-chaining with recursion: Example 2

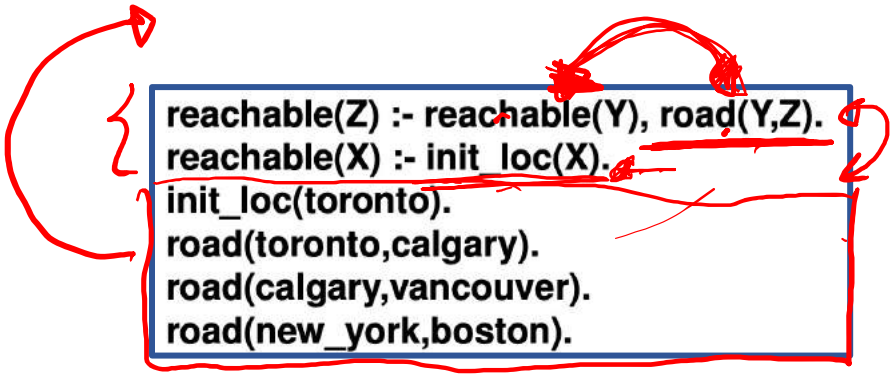



```
1 reachable(Z) :- reachable(Y), road(Y,Z).
2 reachable(X) :- init_loc(X).
3 init_loc(toronto).
4 road(toronto,calgary).
5 road(calgary,vancouver).
6 road(new_york,boston).
```



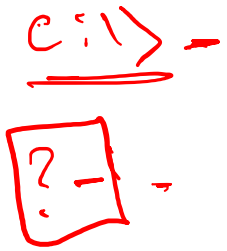
How can we fix example 2?

```
?- reachable(vancouver).
```



1 init_loc(toronto).
2 road(toronto, calgary).
3 road(c, v).
4 road(n, b).
5 reachable(X) :- init_loc(X).
6 reachable(Z) :- road(Y,Z), reachable(Y).

More on Variables



- Some arguments of a predicate may serve as inputs, while others – as outputs.
- PROLOG has an extensive library of predicates. **?-help(predName/arity).**
  - Arity means number of operands
- You may see *predName*(+*X*, –*Y*) that means *X* gets inputs, and *Y* returns outputs.
- Example: *sum*(*X*,*Y*,*Z*) :– *Z* is *X* + *Y*. /*Notation sum(+X,+Y,-Z)*/
  - Syntax for assignment: Variable **is** <some arithmetic expression>.
- **Warning**: if a Var is assigned a value in conjunction, you cannot change it later.
- Query **?- X is 2*2, Y is 5+6, X is Y.** fails
  - since X and Y already assigned
  - **is** works as evaluating equality of two variable (4 not equal to 11). In prolog slag it is equivalent to =:= .

More on Variables

Consider a small *deductive database* for a simplified airline reservation system.

nonStop(FlightNum,C1,C2): *FlightNum* is a non stop flight between cities *C1* and *C2*

```
nonStop(ac103,toronto,montreal).   nonStop(ac103,montreal,ny).
nonStop(ac103,ny,miami).           nonStop(us400,austin,la).
nonStop(us200,miami,la).           nonStop(aa300,ny,austin).
```

dtime(FlightNumber,City,Time): The departure time of *FlightNumber* from *City* is *Time*; time is based on a 24 clock.

```
dtime(ac103,toronto,635).   dtime(ac103,montreal,800).
dtime(aa300,ny,1200).       dtime(us400,austin,1400).
```

atime(FlightNumber,City,Time): The arrival time of *FlightNumber* to *City* is *Time*.

```
atime(ac103,montreal,735).   atime(ac103,ny,930).
atime(us200,la,1800).        atime(aa300,austin,1600).
```

reserved(Person,FlightNumber,City1,City2,Day,Month,Year): *Person* has a reservation on *FlightNumber* on the non-stop flight from *City1* to *City2* on *Day* of *Month* of *Year*.

```
reserved(mary,ac103,toronto,montreal,20,4,2019).
reserved(mary,ac103,montreal,ny,20,4,2019).
reserved(mary,ac103,ny,miami,20,4,2019).
reserved(john,aa300,ny,austin,1,10,2019).
```

Simplifying Assumptions

1) All flights have the same daily schedule, seven days per week. That's why we do not have predicates like

dtime(FlightNumber,City,DayOfWeek,Time)
atime(FlightNumber,City,DayOfWeek,Time).

2) All flights begin and terminate within the same day. This means that there are no flights which depart in the evening, and arrive at their destination during the morning of the following day.

3) Time is measured w.r.t. "universal" time zone, so if it is now 1435 in Toronto, it is also 1435 in Vancouver.

The task is to write additional rules that implement new predicates. They will provide extra reasoning services on top of DB retrieval queries.

Think about the Pearson's airport flights recorded as atomic statements. The new rules faciliate reasoning over this KB.

Simplifying Assumptions

1) All flights have the same daily schedule, seven days per week. That's why we do not have predicates like

dtime(FlightNumber,City,DayOfWeek,Time)
atime(FlightNumber,City,DayOfWeek,Time).

2) All flights begin and terminate within the same day. This means that there are no flights which depart in the evening, and arrive at their destination during the morning of the following day.

3) Time is measured w.r.t. "universal" time zone, so if it is now 1435 in Toronto, it is also 1435 in Vancouver.

The task is to write additional rules that implement new predicates. They will provide extra reasoning services on top of DB retrieval queries.

Think about the Pearson's airport flights recorded as atomic statements. The new rules faciliate reasoning over this KB.

possibleConnectingFlight(Flight1,Flight2,City)

possibleConnectingFlight(Flight1,Flight2,City):
It is possible for a passenger to arrive in *City* on *Flight1* with enough time to catch the departing *Flight2*. Make the simplifying assumption that there is enough time to make the connection if the arrival time of *Flight1* is 50 min less than the departure time of *Flight2*.

```
possibleConnectingFlight(F1,F2,C) :-
```

steps: ① get the arrival time of Flight 1 → T_1

② get the departure time of flight 2 → T_2

③ $T_1 < T_2$

④ $T_2 - T_1 >= 50$ min

Convert into minutes first.

possibleConnectingFlight(Flight1,Flight2,City)

possibleConnectingFlight(Flight1,Flight2,City):
It is possible for a passenger to arrive in *City* on *Flight1* with enough time to catch the departing *Flight2*. Make the simplifying assumption that there is enough time to make the connection if the arrival time of *Flight1* is 50 min less than the departure time of *Flight2*.

```
possibleConnectingFlight(Flight1,Flight2,City):-
    atime(Flight1,City,Time1),
    dtime(Flight2,City,Time2),
    Time1 < Time2,
    diff(Time1,Time2,Minutes),
    Minutes >= 50.

diff(T1,T2,M) :- Hours1 is T1 // 100,  Hours2 is T2 // 100,
    Minutes1 is (T1 mod 100), Minutes2 is (T2 mod 100),
    M is (Hours2 - Hours1)*60 + Minutes2 - Minutes1.
```

Hours1 is 5 x 2,

9:00 // 100 = 9

9:55 // 100 = 9

9:00 mod 100 = 0

9:55 mod 100 = 55

impossibleSchedule(Flight,City)

103 7:30
103 6:30

impossibleSchedule(Flight,City):
The airline schedule is in error in the sense that the arrival time of *Flight* in *City* is greater than its departure time from *City*.

```
impossibleSchedule(F,C) :-
```

steps: ① get arrival time of F in city C → T_1

② get departure time of F in city C → T_2

③ $T_2 < T_1$

impossibleSchedule(Flight,City)

impossibleSchedule(Flight,City):
The airline schedule is in error in the sense that the arrival time of *Flight* in *City* is greater than its departure time from *City*.

```
impossibleSchedule(Flight,City) :- atime(Flight,City,T1),
          dtime(Flight,City,T2),
          T2 < T1.
```

conflictingReservation(Person)

conflictingReservation(Person):
Person has two conflicting reservations, meaning that *Person* has reservations on two different flights which overlap in time, i.e. the two flights are on the same date and the departure time of one of the flights lies between the departure time and arrival time of the other flight.

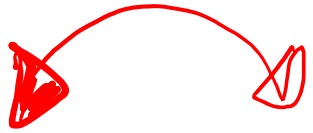
```
conflictingReservation(P) :-
```

steps: (1) find (1) reservation for person

(2) find a second reservation on the day but diff Flight from (1)

(3) Check the arrival and departure of (1) and (2) to find if conflict.

conflictingReservation(Person)



conflictingReservation(Person):
Person has two conflicting reservations, meaning that *Person* has reservations on two different flights which overlap in time, i.e. the two flights are on the same date and the departure time of one of the flights lies between the departure time and arrival time of the other flight.

```
conflictingReservation(Person) :-
    reserved(Person,Flight,City1,City2,Day,Month,Year),
    reserved(Person,F,C1,C2,Day,Month,Year),
    not Flight = F,
    dtime(Flight,City1,Time1),
    atime(Flight,City2,Time2),
    dtime(F,C1,T),
    Time1 < T, T < Time2.
```

twoFlightConnection(Flight1,Flight2,City1,City,City2)

twoFlightConnection(Flight1,Flight2,City1,C,City2):
Flight1 is a direct flight from *City1* to *C*, and *Flight2* is a different direct flight from *C* to *City2*. In other words, one can fly from *City1* to *City2* via *C*, changing airplanes exactly once in *C*. Moreover, *Flight2* must be a *possibleConnectingFlight* for *Flight1* in *C*. For the above example, *twoFlightConnection(ac103,aa300,toronto,ny,austin)* is true, whereas the following two are both false (for different reasons):

twoFlightConnection(ac103,aa300,toronto,montreal,austin)
twoFlightConnection(aa300,us400,ny,austin,la)

```
twoFlightConnection(F1,F2,City1,C,City2) :-
```

twoFlightConnection(Flight1,Flight2,City1,City,City2)

twoFlightConnection(Flight1,Flight2,City1,C,City2):
Flight1 is a direct flight from *City1* to *C*, and *Flight2* is a different direct flight from *C* to *City2*. In other words, one can fly from *City1* to *City2* via *C*, changing airplanes exactly once in *C*. Moreover, *Flight2* must be a *possibleConnectingFlight* for *Flight1* in *C*. For the above example, *twoFlightConnection(ac103,aa300,toronto,ny,austin)* is true, whereas the following two are both false (for different reasons):

twoFlightConnection(ac103,aa300,toronto,montreal,austin)
twoFlightConnection(aa300,us400,ny,austin,la)

```
twoFlightConnection(F1,F2,C1,C,C2) :-
    directFlight(F1,C1,C),
    directFlight(F2,C,C2),
    not F1 = F2,
    possibleConnectingFlight(F1,F2,C).
```

CPS721, Sections 051 to 091
Lecture 6

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Lists in Prolog**

A recurring problem in Prolog

A problem that comes up again and again in Prolog is that we have to write collections of predicates that are very similar *except* for the number of arguments they take. For example,

```
uniq_person3(X,Y,Z) :- ...    for 3 people
uniq_person4(X,Y,Z,U) :- ...    for 4 people
uniq_person5(X,Y,Z,U,V) :- ...    for 5 people
```

What would be much clearer and convenient, is if we could write just *one* such predicate

```
uniq_people(L) :- ...    for any number of people
```

that would work for any *collection* `L` of people, no matter how big or small.

Lists

A list is a sequence of objects which are called the *elements* of the list.
Here are some examples:

- `[anna, karenina]`
  A two element list, whose first element is `anna` and whose second element is `karenina`.
- `[intro, to, programming, in, prolog]`
  A five element list.
- `[1, 2, 3, 3, 2, 1]`
  Lists may have repeated elements.
- `[]`
  A list with no elements. The *empty* list.

Structured lists

Lists may contain other lists as elements:

- `[[john, 23], [mary, 14], [hello]]`
  A list whose three elements are also lists.
- `[1, john, ['199y', john]]`
  Another three element list.
- `[[]]`
  A one element list, whose element is the empty list.

Note: A one element list is different from the element itself

`[anna]` is different from `anna`;
`[[]]` is different from `[]`.

Heads and tails

The first element of a non-empty list is called the *head* of the list.

The rest of the non-empty list is called the *tail*.

For example,

- `[a, b, c, d]` has head `a` and tail `[b, c, d]`;
- `[[a], b, [c]]` has head `[a]` and tail `[b, [c]]`;
- `[a, [b, c]]` has head `a` and tail `[[b, c]]`;
- `[[a, b]]` has head `[a, b]` and tail `[]`;
- `[]` has neither head not tail.

Note that the head of a list can be *anything*, but the tail is always a *list*.

List as Prolog terms

Prolog terms are now defined as: constants, variables, numbers, *or lists*.

Lists can be written in one of two ways:

1. a *left square parenthesis*, followed by a sequence of terms separated by *commas*, and terminated by a *right square parenthesis*;

   for example: `[1, -4, Z, [a, X, 'b 2'], joe]`

2. a *left square parenthesis*, followed by a non-empty sequence of terms separated by *commas*, followed by a *vertical bar*, followed by a term that denotes a list, and terminated by a *right square parenthesis*.

   for example: `[1, X, 3]` can be written

   `[1 | [X, 3]]` or

   `[1, X | [3]]` or

   `[1, X, 3 | []]`.

Unification with lists

Since variables can appear within lists, we need to be concerned about which pair of lists *unify* (as needed for back-chaining).

The basic idea is this:

- two lists without variables match when they are identical, element for element;
- two lists with distinct variables match when the variables can be given values that make the two lists identical, element for element.

We will see a number of examples of matching lists on the next slides, but the idea can be induced from the example below:

the atom `p([X, 3 | Z])` unifies with `p([b, Y, c, b])`
for `X=b, Y=3, Z=[c, b]`.

Do these lists match (1)?

`[]` and `[]`

`[a,b,c]` and `[a,b,c]`

`[X]` and `[a]` with `X=a`

`[a,b,X]` and `[Y,b,Y]` with `X=a Y=a`

`[X,X]` and `[[Y,a,c],[b,a,c]]` with `X=[b,a,c] Y=b`

`[[X]]` and `[Y]` with `X=_G23 Y=[_G23]`

`[a,b,c]` and `[a|[b,c]]`

`[a,b,c]` and `[a|[b|[c]]]`

`[X|Y]` and `[a]` with `X=a Y=[]`

`[a|[b,c]]` and `[a,X,Y]` with `X=b Y=c`

`[a,b,c]` and `[X|Y]` with `X=a Y=[b,c]`

`[X,Y|Y]` and `[a,[b,c],b,c]` with `X=a Y=[b,c]`

Do these lists match (2)?

Lists do not match if they have different numbers of elements or if at least one corresponding element does not match.

`[a]` and `[]`

`[]` and `[[]]`

`[X,Y]` and `[U,V,W]`

`[a,b,c]` and `[X,b,X]`

`[X|Y]` and `[]`

Observe: `X` matches anything, including any list

`[X]` matches any list with exactly one element

`[X|Y]` matches any list with at least one element

`[X,Y]` matches any list with exactly two elements

Writing programs that use lists

Programs that use lists usually end up being *recursive* since we may not know in advance how big a list we will need to deal with.

We do know that each list will have *some number* k of elements.

This means that we can work recursively as follows:

- write clauses to handle the base case where $k = 0$
  (that is, for the empty list);
- assume the case for $k = n$ has already been taken care of,
  and write clauses to handle the case where $k = (n + 1)$;

  Assume the list `T` is handled, and write clauses for `[H|T]`.

If we can do this, we will have handled all lists (by mathematical induction).

Now we consider four examples.

1. A list of people (1)

As a first example of a predicate that deals with lists, consider the following:

> Imagine that we already have a predicate `person`(x) that holds when x is a person. We would like to write the clauses for another predicate `person_list`(z) that holds when z is a list whose elements are all people.

This will be a recursive predicate:

- we want clause(s) for the empty list: `person_list([]).`
- assuming we have `person_list(Z)` that we can use, we want clause(s) for a list with one more element: `person_list([P|Z]).`

If we can provide clauses for both of these cases, we are done.

1. A list of people (2)

Here is a definition of `person_list`:

```
% We assume the person(X) predicate is defined elsewhere.
% The empty list is a (trivial) list of persons.
person_list([]).
% If P is person and Z is a list of persons then [P|Z] is a list of persons.
person_list([P|Z]) :- person(P), person_list(Z).
```

This is how the query `person_list([john,joe,sue])` would work:

- `[john,joe,sue]` unifies with `[P|Z]` for `P=john` and `Z=[joe,sue]`;
- `[joe,sue]` unifies with `[P|Z]` for `P=joe` and `Z=[sue]`;
- `[sue]` unifies with `[P|Z]` for `P=sue` and `Z=[]`;
- `[]` unifies with `[]` and succeeds immediately.

Note how the two notations for lists work together on this.

2. Predicate for list membership (1)

Suppose we want a predicate that will tell us whether or not something is an element of a list. In other words, we want this behaviour:

```
?- elem(b,[a,b,c,d]).
Yes
?- elem(f,[a,b,c,d]).
No
```

Most Prolog systems provide a predefined predicate (called `member`) with this behaviour. Nonetheless, we will define our own.

This will be a recursive predicate:

- we write clauses for the empty list

  Nothing to write, since the query `elem(X,[])` should always fail.

- assuming a list `L` is handled, write clauses to handle the list `[H|L]`

  The query `elem(X,[H|L])` should succeed if either `X = H` succeeds or `elem(X,L)` succeeds. So two clauses are needed.

2. Predicate for list membership (2)

Here is how we would define the `elem` predicate in Prolog:

```
% elem(X,L) holds when X is an element of list L
% X is an element of any list whose head is X.
elem(X,[X|_]).
% If X is an element of L, then it is an element of [_|L].
elem(X,[_|L]) :- elem(X,L).
```

Note the use of the *underscore* (anonymous variable):

- in the first clause, we don't care what the tail is;
- in the second, we don't care about the head.

```
?- elem(c,[a,b,c,d,e]).
 T Call: (7) elem(c, [a, b, c, d, e])
 T Call: (8) elem(c, [b, c, d, e])
 T Call: (9) elem(c, [c, d, e])       % Here we get to use the 1st clause
 T Exit: (9) elem(c, [c, d, e])       % which succeeds immediately!
 T Exit: (8) elem(c, [b, c, d, e])
 T Exit: (7) elem(c, [a, b, c, d, e])
```

3. A list of unique people

Using `member` (or `elem`) we can now return to our original motivation:

> write the clauses for a predicate `uniq_people`(z) that holds when z is a list of people (of any size) *that are all different*.

As before, we assume that we have a predicate `person` already defined.

This will be a recursive predicate:

- The empty list `[]` is a (trivial) list of unique people.
- If `L` is a list of unique people, and `P` is a person, and `P` is not an element of `L`, then `[P|L]` is also a list of unique people.

This gives us the following two Prolog clauses:

```
uniq_people([]).
uniq_people([P|L]) :- uniq_people(L), person(P), \+ member(P,L).
```

4. Joining two lists (1)

Suppose we want a list predicate that will join two lists together. In other words, we want this behaviour:

```
?- join([a,b,c,d],[e,f,g],L).
L=[a,b,c,d,e,f,g]
Yes
?- join([],[a,b,c,d],L).
L=[a,b,c,d]
Yes
```

Most Prolog systems provide a predefined predicate (called `append`) with this behaviour. We again define our own.

The predicate will once again be recursive.

However, the predicate `join` needs to work for any *two* lists: the first list can be empty or non-empty, and the second list can also be empty or non-empty.

4. Joining two lists (2)

It is sufficient to do recursion on the *first argument* only:

- write clauses to handle the case where the first argument is `[]` and the second argument is any list `L`;
- write clauses to handle the case where the first argument is `[H|T]` and the second argument is any list `L` (assuming `join` already works when the first argument is `T` and the second argument is `L`).

Here is the program we get:

```
% join(X,Y,Z) means the result of joining X and Y is Z.
% Joining [] and any list L gives L itself.
join([],L,L).
% If joining T and L gives Z, then joining [H|T] and L gives [H|Z].
join([H|T],L,[H|Z]) :- join(T,L,Z).
```

Tracing join

```
?- join([a,b,c,d],[e,f,g],Z).
 T Call: (7) join([a, b, c, d], [e, f, g], _G306)
 T Call: (8) join([b, c, d], [e, f, g], _G378)
 T Call: (9) join([c, d], [e, f, g], _G381)
 T Call: (10) join([d], [e, f, g], _G384)
 T Call: (11) join([], [e, f, g], _G387)          % Here we get to use
 T Exit: (11) join([], [e, f, g], [e, f, g])      % the first clause
 T Exit: (10) join([d], [e, f, g], [d, e, f, g])
 T Exit: (9) join([c, d], [e, f, g], [c, d, e, f, g])
 T Exit: (8) join([b, c, d], [e, f, g], [b, c, d, e, f, g])
 T Exit: (7) join([a, b, c, d], [e, f, g], [a, b, c, d, e, f, g])
Z = [a, b, c, d, e, f, g]
Yes
```

Note that the first argument will be progressively reduced until it becomes `[]`.

At this point, the third argument must be equal to the second.

Then, each recursive query gets a turn to put a new head onto the third argument.

(Note the use of `H` in the head of the clause.)

CPS721, Sections 051 to 091
Lecture 7

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Lists in Prolog**

Writing programs that use lists

Programs that use lists usually end up being *recursive* since we may not know in advance how big a list we will need to deal with.

We do know that each list will have *some number* k of elements.

This means that we can work recursively as follows:

- write clauses to handle the base case where $k = 0$ (that is, for the empty list);
- assume the case for $k = n$ has already been taken care of, and write clauses to handle the case where $k = (n + 1)$;

  Assume the list `T` is handled, and write clauses for `[H|T]`.

If we can do this, we will have handled all lists (by mathematical induction).

Now we consider four examples.

1. A list of people (1)

As a first example of a predicate that deals with lists, consider the following:

> Imagine that we already have a predicate `person`(x) that holds when x is a person. We would like to write the clauses for another predicate `person_list`(z) that holds when z is a list whose elements are all people.

person(3) → No
person(bob) → Yes

→ [] already verified

1. A list of people (2)

Here is a definition of `person_list`:

```
% We assume the person(X) predicate is defined elsewhere.
% The empty list is a (trivial) list of persons.
person_list([]).
% If P is person and Z is a list of persons then [P|Z] is a list of persons.
person_list([P|Z]) :- person(P), person_list(Z).
```

This is how the query `person_list([john,joe,sue])` would work:

- `[john,joe,sue]` unifies with `[P|Z]` for `P=john` and `Z=[joe,sue]`;
- `[joe,sue]` unifies with `[P|Z]` for `P=joe` and `Z=[sue]`;
- `[sue]` unifies with `[P|Z]` for P=sue and `Z=[]`;
- `[]` unifies with `[]` and succeeds immediately.

Note how the two notations for lists work together on this.

2. Predicate for list membership (1)

Suppose we want a predicate that will tell us whether or not something is an element of a list. In other words, we want this behaviour:

```
?- elem(b,[a,b,c,d]).
Yes
?- elem(f,[a,b,c,d]).
No
```

Most Prolog systems provide a predefined predicate (called `member`) with this behaviour. Nonetheless, we will define our own.

2. Predicate for list membership (2)

Here is how we would define the `elem` predicate in Prolog:

```
% elem(X,L) holds when X is an element of list L
% X is an element of any list whose head is X.
elem(X,[X|_]).
% If X is an element of L, then it is an element of [_|L].
elem(X,[_|L]) :- elem(X,L).
```

Note the use of the *underscore* (anonymous variable):

- in the first clause, we don't care what the tail is;
- in the second, we don't care about the head.

```
?- elem(c,[a,b,c,d,e]).
 T Call: (7) elem(c, [a, b, c, d, e])
 T Call: (8) elem(c, [b, c, d, e])
 T Call: (9) elem(c, [c, d, e])
 T Exit: (9) elem(c, [c, d, e])
 T Exit: (8) elem(c, [b, c, d, e])
 T Exit: (7) elem(c, [a, b, c, d, e])
```

```
% Here we get to use the 1st clause
% which succeeds immediately!
```

3. A list of unique people

Using `member` (or `elem`) we can now return to our original motivation:

> write the clauses for a predicate `uniq_people(`z`)` that holds when z is a list of people (of any size) *that are all different*.

As before, we assume that we have a predicate `person` already defined.

4. Joining two lists (1)

Suppose we want a list predicate that will join two lists together. In other words, we want this behaviour:

```
?- join([a,b,c,d],[e,f,g],L).
L=[a,b,c,d,e,f,g]
Yes

?- join([],[a,b,c,d],L).
L=[a,b,c,d]
Yes
```

Most Prolog systems provide a predefined predicate (called `append`) with this behaviour. We again define our own.

The predicate will once again be recursive.

However, the predicate `join` needs to work for any *two* lists: the first list can be empty or non-empty, and the second list can also be empty or non-empty.

4. Joining two lists (2)

It is sufficient to do recursion on the *first argument* only:

- write clauses to handle the case where the first argument is `[]` and the second argument is any list `L`;
- write clauses to handle the case where the first argument is `[H|T]` and the second argument is any list `L` (assuming `join` already works when the first argument is `T` and the second argument is `L`).

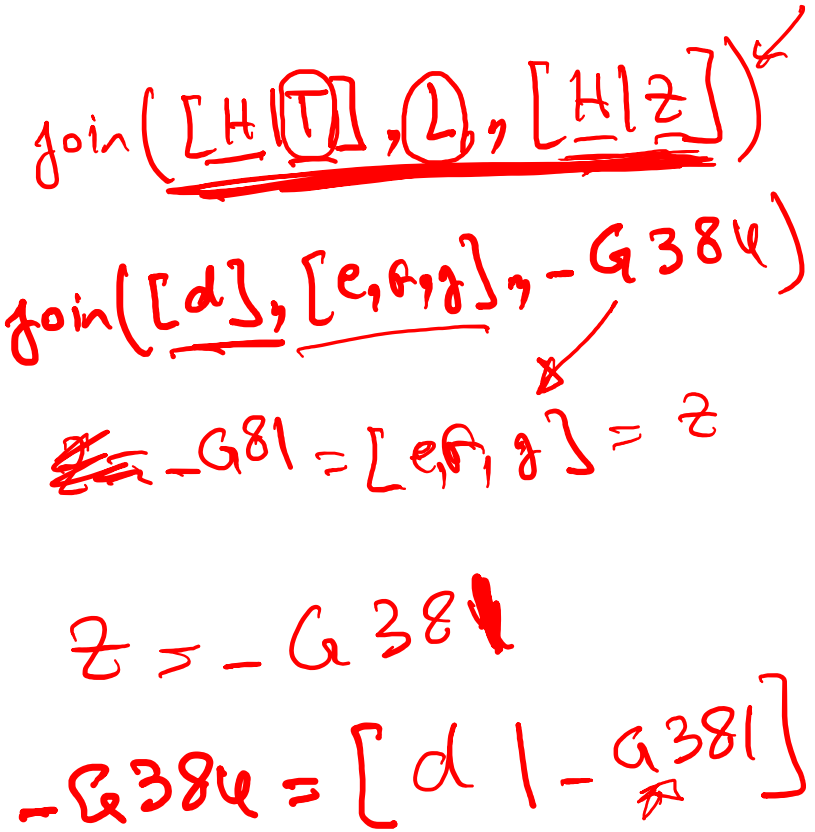
Here is the program we get:

```
% join(X,Y,Z) means the result of joining X and Y is Z.
% Joining [] and any list L gives L itself.
join([],L,L).
% If joining T and L gives Z, then joining [H|T] and L gives [H|Z].
join([H|T],L,[H|Z]) :- join(T,L,Z).
```

Tracing join

```
?- join([a,b,c,d],[e,f,g],Z).
 T Call: (7) join([a, b, c, d], [e, f, g], _G306)
 T Call: (8) join([b, c, d], [e, f, g], _G378)
 T Call: (9) join([c, d], [e, f, g], _G381)
 T Call: (10) join([d], [e, f, g], _G384)
 T Call: (11) join([], [e, f, g], _G387)
 T Exit: (11) join([], [e, f, g], [e, f, g])
 T Exit: (10) join([d], [e, f, g], [d, e, f, g])
 T Exit: (9) join([c, d], [e, f, g], [c, d, e, f, g])
 T Exit: (8) join([b, c, d], [e, f, g], [b, c, d, e, f, g])
 T Exit: (7) join([a, b, c, d], [e, f, g], [a, b, c, d, e, f, g])
Z = [a, b, c, d, e, f, g]
Yes
```

```
% Here we get to use
% the first clause
```



Note that the first argument will be progressively reduced until it becomes `[]`.

At this point, the third argument must be equal to the second.

Then, each recursive query gets a turn to put a new head onto the third argument.
(Note the use of `H` in the head of the clause.)

Using member function

In addition to *testing* for membership in a list, the `member` predicate (or our `elem` predicate) can also be used to *generate* the elements of a list.

```
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
No
```

Generate finite sets only

Using a predicate to generate a set of candidates that we then test with other predicates is a powerful feature that we will use extensively in this course.

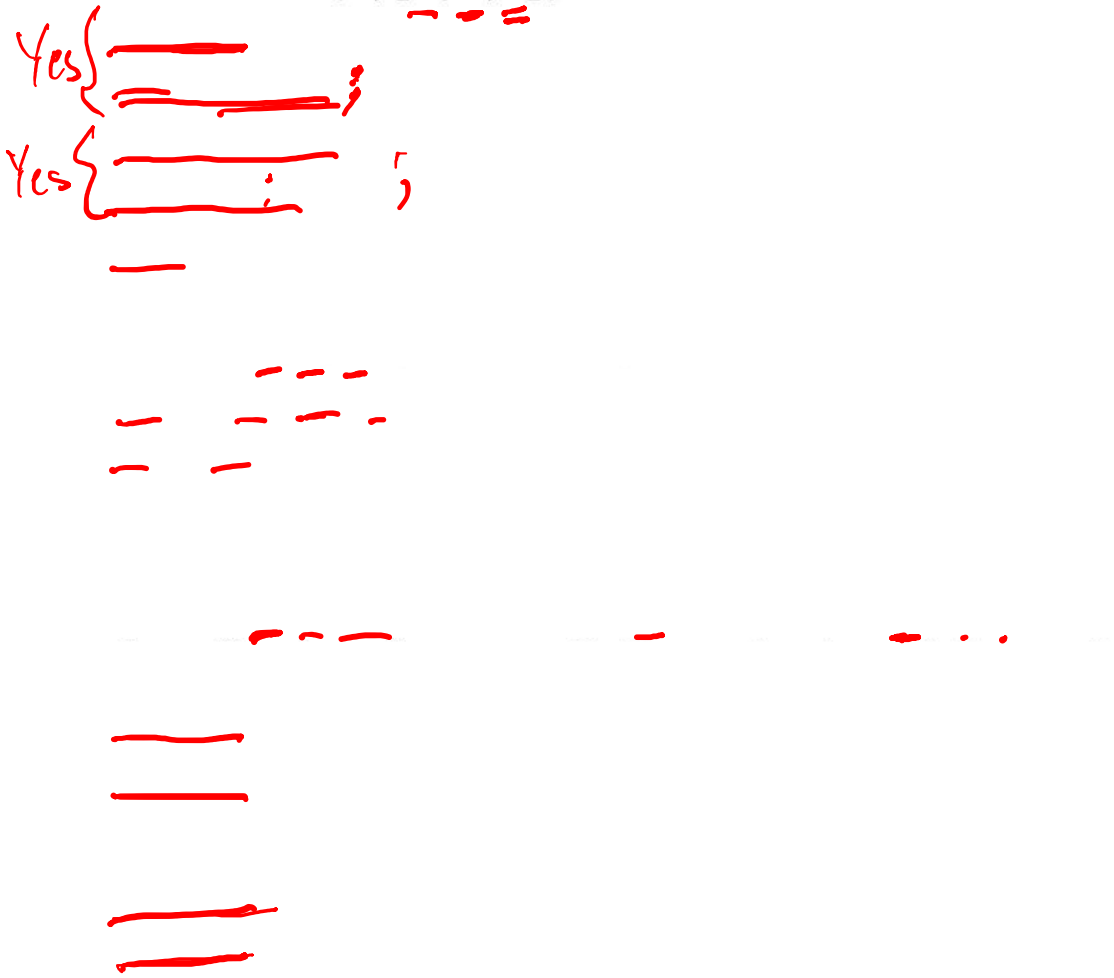
However, some care is required to make sure that predicates used this way will not generate an *infinite* set of candidates.

For example,

```
?- member(3,L).          % what are lists that contain 3 as an element?
L = [3|_G214] ;                  % any list whose head is 3
L = [_G213, 3|_G217] ;           % any list whose 2nd element is 3
L = [_G213, _G216, 3|_G220] ;    % and so on
yes

?- member(3,L), a=b.
ERROR: (user://1:71):            % instead of just failing, the query
        Out of global stack      % tries to run forever!
```

More membership queries

```
?- member(a,[X,b,Y]).
```



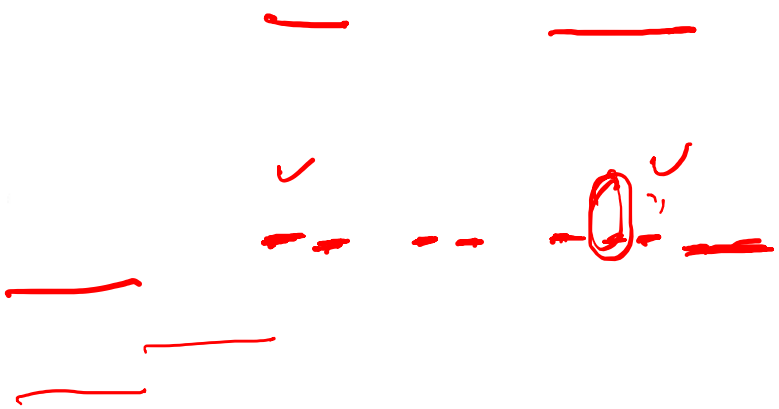
Generating lists with append

```
?- append(X,Y,[a,b,c]).
X = []
Y = [a, b, c] ;
X = [a]
Y = [b, c] ;
X = [a, b]
Y = [c] ;
X = [a, b, c]
Y = [] ;
No
```

```
% What pairs of lists when joined give [a,b,c]?
```

append ([x_1, x_2] , [y_1, y_2] , z)

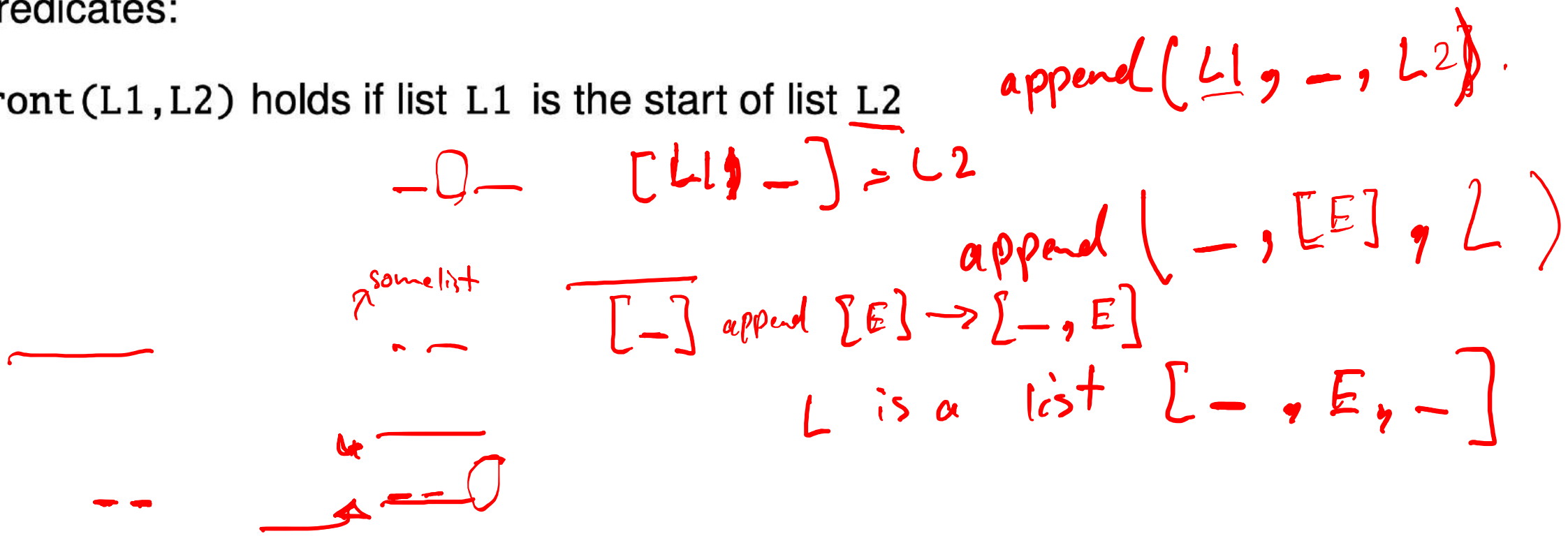
z = [x_1, x_2, y_1, y_2]



Defining new predicates using append

Using `append` (or our defined version, `join`), it possible to define several new predicates:

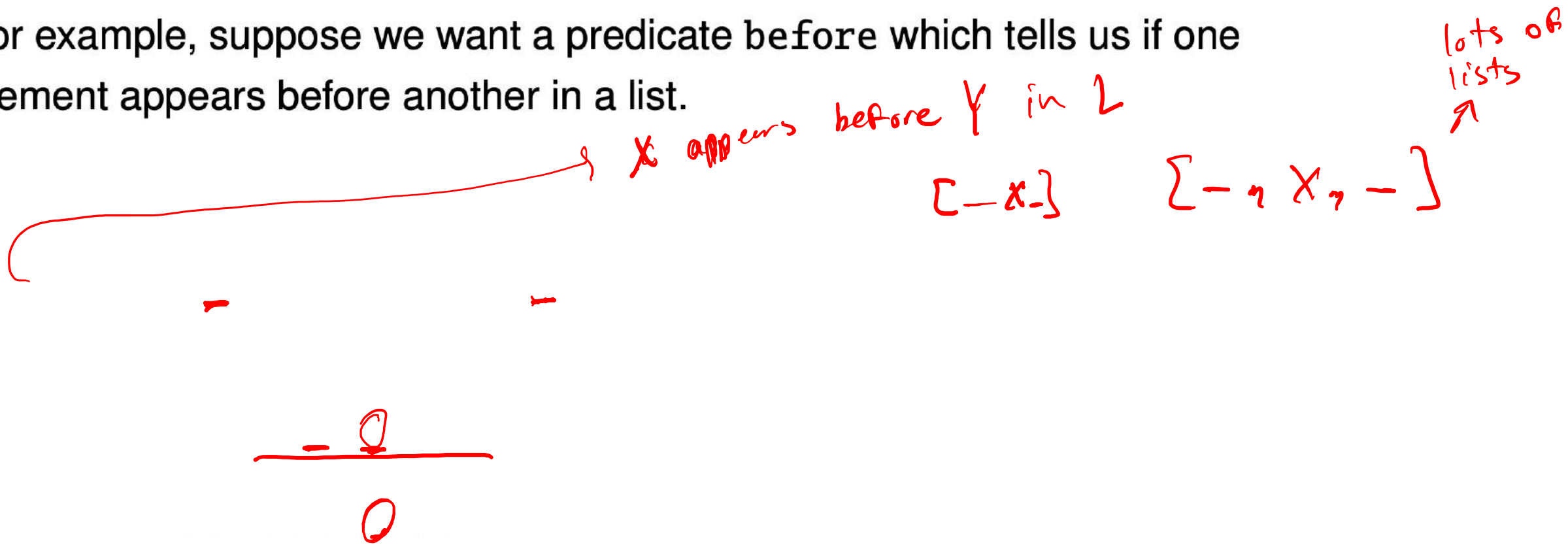
- `front(L1,L2)` holds if list `L1` is the start of list `L2`



Generating finite sets with append

As with the `member` predicate, we are allowed to use variables freely, but we must be careful not to generate infinite sets of candidate lists.

For example, suppose we want a predicate `before` which tells us if one element appears before another in a list.



The problem with the before predicate

The trouble is that the first definition *generates* a list `Z` that contains `X`, and then does other things.

But there are infinitely many such lists!

So if the second part of the query fails, it can run forever (or until the lists are big enough to exhaust all the memory).

Solution: generate the `Z` as a sublist of the given `L` first.

```
% A better solution
before(X,Y,L) :- append(Z,[Y|_],L), append(_,[X|_],Z).
```

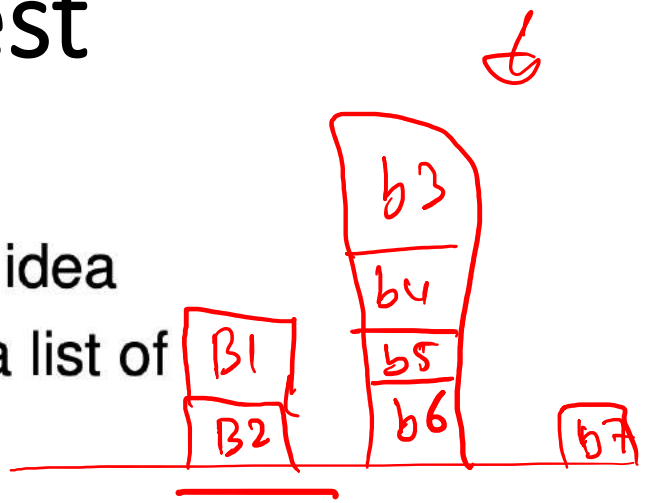
This gives us the following:

```
?- before(2,4,[1,2,8,4,3,7]).
Yes

?- before(2,4,[1,2,8,9,3,7]).
No
```

The blocks world using generate and test

From now on, we will use `member` and `append` freely.

Here, for example, is a redone version of the blocks world, using the idea that a *scene* is a list of stacks in left-to-right order, where a *stack* is a list of blocks in top-to-bottom order.




```
% This is a list-based version of the blocks-world program.

% X appears before Y in list L.
before(X,Y,L) :- append(Z,[Y|_],L), append(_,[X|_],Z).

% The given blocks-world scene: three stacks of blocks
scene([[b1,b2],[b3,b4,b5,b6],[b7]]).

% above(X,Y) means that block X is somewhere above block Y.
above(X,Y) :- scene(L), member(Stack,L), before(X,Y,Stack).

% left(X,Y) means that block X is somewhere left of block Y.
left(X,Y) :- scene(L), before(Stack1,Stack2,L),
             member(X,Stack1), member(Y,Stack2).

right(Y,X) :- left(X,Y).
```

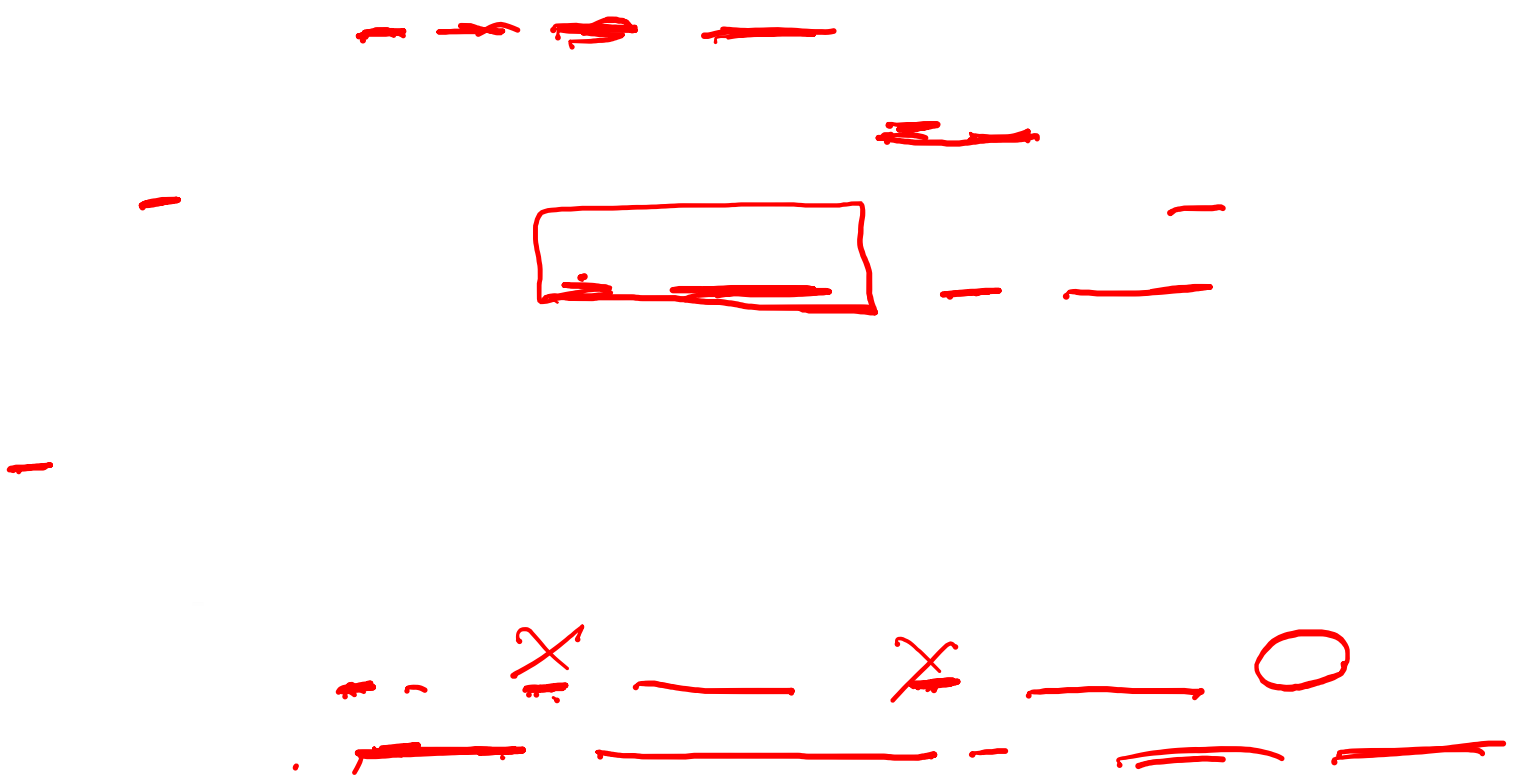
gets L from scene

The predicate replaceFirst (X , Y , L1, L2)

"The list *L2* results from replacing the very first occurrence of *X* in the list *L1* by *Y*"

Encode this English sentence with the predicate *replaceFirst*$(+X, +Y, +L1, -L2)$.

Do case analysis: there are 3 cases when the sentence is true. What are the cases ?



Examples of replaceFirst (X , Y , L1, L2)

```
1 replaceFirst(X, Y, [ ], [ ]).
2 replaceFirst(X, Y, [X | L], [Y | L]).
3 replaceFirst(X,Y, [Z | L1], [Z | L2]) :-
                    not X = Z, replaceFirst(X,Y, L1, L2).
```

Examples: A. Transform "house" into "mouse"

?- replaceFirst(h, m, [h,o,u,s,e], L).

```
L = [m,o,u,s,e]
Yes (0.00s cpu, solution 1, maybe more) ;
No (0.02s cpu)
```

B. Transform `[s,a,n,d]` into `[g,o,l,d]`. Exercise.

Hint: call *replaceFirst* consecutively several times: *s-> g*, *a -> o*, *n -> l*.

The predicate replaceAll (X , Y , L1, L2)

"The list *L2* results from replacing all occurrences of *X* in the list *L1* by *Y*"

Encode this English sentence by the predicate *replaceAll*(*X*, *Y*, *L1*, *L2*)

Develop code for *replaceAll* from our Prolog code for *replaceFirst*. How?

The predicate sum(L, S)

The predicate *sum*(*List*, *S*) is true if *S* is the sum of the numbers in *List*.
Do case analysis, but explore a few simple cases before arriving at correct recursion.

The predicate length(List,N)

The predicate *length*(*List*, *N*) is true if *N* is the number of elements in *List*.
Do case analysis, but explore a few simple cases before arriving at correct recursion.

CPS721, Sections 051 to 091
Lecture 8

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Lists in Prolog**
- **Terms**

The predicate replaceAll (X , Y , L1, L2)

"The list *L2* results from replacing all occurrences of *X* in the list *L*1 by *Y*"

Encode this English sentence by the predicate *replaceAll*(*X*, *Y*, *L*1, *L2*)

Develop code for *replaceAll* from our Prolog code for *replaceFirst*. How?

```
replaceFirst(X, Y, [ ], [ ]).                    /* how to change this line?*/
```

replaceAll(X, Y, [ ], [ ]).

```
replaceFirst(X, Y, [X | Tail], [Y | Tail]).   /* what to chamge here?*/
```

replaceAll(X, Y, [X | Tail1], [Y | Tail2]) :-
replaceAll(X, Y, Tail1, Tail2).

```
replaceFirst(X,Y, [Z | L1], [Z | L2]) :-
                    not X = Z, replaceFirst(X,Y, L1, L2).
```

How to revise this last rule?

replaceAll(X, Y, [Z | L1], [Z | L2]) :-
not X = Z, replaceAll(X, Y, L1, L2).

Example. What list is the result of replacing *p* by *m* everywhere in the list [*p*, *a*, *p*, *a*] ?

?- replaceAll(p, m, [p,a,p,a], L).

```
L = [m,a,m,a].
```

Transform `[s,p,e,e,d]` into `[p,i,z,z,a]`. Exercise.

The predicate sum(L, S)

The predicate *sum*(*List*, *S*) is true if *S* is the sum of the numbers in *List*.
Do case analysis, but explore a few simple cases before arriving at correct recursion.

If the input is the empty list, then the sum of its elements is 0.

```
sum([ ], 0).
```
(*)

If the input list has only 1 element *X*1, then the sum is simply *X*1.

```
sum([X1], X1).
```

If the input list has only 2 elements *X*1 and *X*2, then the sum is $X1 + X2$.

```
sum([X1,X2], S) :- S is X1+X2.
```

If the input list has only 3 elements [*X*1, *X*2, *X*3], then to compute sum, add *X*3 to the sum of *X*1 and *X*2.

```
sum([X1,X2,X3], S) :- sum([X1,X2],M), S is X3+M.
```

For an arbitrary list, compute recursively sum of the tail, and then add the number in the head of input to the intermediate sum.

```
sum([Head | Tail], S) :- sum(Tail,M), S is Head + M.
```
(**)

Order of computation in this rule is important ! We cannot do addition before the recursive call.
Can remove analysis, and keep only (*) and (**).

The predicate length(List,N)

The predicate *length*(*List*, *N*) is true if *N* is the number of elements in *List*.
Do case analysis, but explore a few simple cases before arriving at correct recursion.

If the input is the empty list, then its length is 0.

```
length([ ], 0).
```
(*)

If the input list has only one element *X*1, then its length is simply 1.

```
length([X1], 1).
```

If the input list has only 2 elements *X*1 and *X*2, then the length is 2.

```
length([X1,X2], L) :- L is 2.
```

If the input list has only 3 elements [*X*1, *X*2, *X*3], then to compute its length, add 1 to the length of [*X*1, *X*2].

```
length([X1,X2,X3], L) :- length([X1,X2],M), L is M+1.
```

For an arbitrary non-empty list, compute recursively the length of its tail, then add 1.

```
length([Head | Tail], L) :- length(Tail,M), L is M+1.
```
(**)

Order of computation in this rule is important ! We cannot do addition before the recursive call.
Can remove analysis, and keep only (*) and (**).

Predicates vs Terms

In discrete mathematics you studied relations and function. Similarly, in logic we have

predicates – logical statements that represent relations between individuals. Each predicate can be true or false, depending on its arguments. Example: $parent(Human, P)$ is a predicate. Each human has 2 parents, a human is in relation with parents.

terms – functions that map given individual(s) into a unique individual in a domain. Each function may take different values depending on the arguments. Example: $mother(Human) = M$. Each human has only one mother, this is a functional dependence.

Terms correspond to structures in other programming languages. They help to represent structured data as a single unit.

Note that atomic statements and rules can use predicates only, but inside predicates we can use arguments composed from terms.

Terms can be used only as arguments of predicates or as arguments of $=$.

Example: Families KB

To illustrate terms we consider a database with structured information about people and families. We introduce the following.

The predicate *family*(*Husband*, *Wife*, *ListOfChildren*) is true if the 1st and the 2nd argument represent a father and a mother of children in the 3rd argument

Two terms:

person(*FirstName*, *LastName*, *DOB*, *Job*)

date(*Day*, *Month*, *Year*)

These terms can be used as arguments of the predicate *family* to represent information in a more structured way.

Since *person* and *date* are terms, they can only occur inside the predicate *family* or as arguments of “equality" ($=$).

Example: consider the predicate “less than" and a term composed from the function “sum", both written with the usual notation, e.g.,

$$(a + b) + c \leq (b + c)$$

Notice terms are arguments of the predicate “$\leq$". But an arithmetical expression $(b + c)$ by itself would not have a truth value.

Two atomic statements on KB

To be more specific, let us consider a couple of atomic statements using the predicate *family*.

family(
person(*tom*, *fox*, *date*(7, *may*, 1992), *ibm*),
person(*ann*, *fox*, *date*(9, *may*, 1994), *microsoft*),
[*person*(*pat*, *fox*, *date*(5, *august*, 2017), *unemployed*),
person(*jim*, *fox*, *date*(5, *august*, 2017), *unemployed*)]).

family(
person(*john*, *nixon*, *date*(29, *february*, 1996), *unemployed*),
person(*mary*, *nixon*, *date*(15, *august*, 1997), *unemployed*),
[ ]).

And so on... Given this collection of atomic statements about families, we can retrieve information in a structured way using Prolog queries.

Pay attention how we use *person* and *date* inside *family*

Queries with terms

For the sake of simplicity, let us assume that a wife and a husband have the same family name. Recall the arguments:

family(*Husband*, *Wife*, *ListOfChildren*)
person(*FirstName*, *LastName*, *DOB*, *Job*)
date(*Day*, *Month*, *Year*)

Formulate a query in Prolog that retrieves information about all families with the last name "*armstrong*" such that the husband was born after 1977.

?- *family*(*person*(*Name*, *armstrong*, *date*(*D*, *M*, *Y*), *J*), *W*, *L*), $Y > 1977$.

Formulate a query in Prolog to find information about all employed women who have at least 2 children. Assume that all people without a job have the constant *unemployed* as the 4th argument of their term *person*.

?- *family*(*H*, *person*(*FirstN*, *LastN*, *D*, *J*), [*Ch*1, *Ch*2|*List*]), *not* $J = unemployed$.

Defining new predicates using family predicate

We can easily define several new useful predicates using the predicate *family*. To do this, it is sufficient to remember the meaning of its arguments. In the following questions, you have to write Prolog rules in terms of *family*:

- $husband(X) :- ....$
- $wife(X) :- ....$
- $child(X) :- ....$
- $human(X) :- ....$
- $dateofbirth(X, Day, Month, Year) :- ...$

$husband(X) :- family(X, M, L).$

$wife(X) :- family(H, X, L).$

$child(X) :- family(H, W, Children), member(X, Children).$

$human(X) :- husband(X).\ \ human(X) :- wife(X).\ \ human(X) :- child(X).$

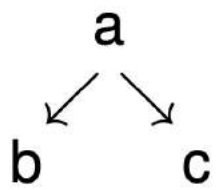
$$dateofbirth(X, Day, Month, Year) :- human(X), X = person(FN, LN, date(Day, Month, Year), Job).$$

Binary Trees as Terms

We can use term with 3 arguments

```
tree(Element,LeftBranch,RightBranch)
```

to represent a binary tree. The empty tree can be represented by the constant `void` We represent the tree



as `tree(a, tree(b, void, void), tree(c,void,void))`

It is important to realize that binary tree is a recursive data structure: *a sub-tree of a tree is a tree itself*.

If we consider the root of the tree, then its left child is a root of a left sub-tree, which is also a tree. Similarly, its right child is a root of a right sub-tree, which is also a binary tree.

Consider a few recursive programs take advantage of recursiveness of binary trees.

As usual, we do case analysis to write recursive programs.

Recursion over terms

The predicate *binaryTree*(*X*) is true if *X* is a binary tree.
What is the base case ? What is a recursive case ?

```
binaryTree(void).

binaryTree( tree(Element, Left, Right) ) :-
          binaryTree(Left), binaryTree(Right).
```

The predicate *memberOfTree*(*Element*, *T*) is true if *Element* is a member of a binary tree *T*. To implement this predicate do case analysis.

```
/* memberOfTree(Element, tree(Root,Left,Right) ) :- Element=Root.*/
memberOfTree(Element, tree(Element,Left,Right) ).

memberOfTree(Element, tree(Node,Left,Right) ) :-
          memberOfTree(Element,Left).

memberOfTree(Element, tree(Node,Left,Right) ) :-
          memberOfTree(Element,Right).
```

The predicate sum(L, S)

The predicate *sum*(*List*, *S*) is true if *S* is the sum of the numbers in *List*.
Do case analysis, but explore a few simple cases before arriving at correct recursion.

If the input is the empty list, then the sum of its elements is 0.

```
sum([ ], 0).
```
(*)

If the input list has only 1 element *X*1, then the sum is simply *X*1.

```
sum([X1], X1).
```

If the input list has only 2 elements *X*1 and *X*2, then the sum is $X1 + X2$.

```
sum([X1,X2], S) :- S is X1+X2.
```

If the input list has only 3 elements [*X*1, *X*2, *X*3], then to compute sum, add *X*3 to the sum of *X*1 and *X*2.

```
sum([X1,X2,X3], S) :- sum([X1,X2],M), S is X3+M.
```

For an arbitrary list, compute recursively sum of the tail, and then add the number in the head of input to the intermediate sum.

```
sum([Head | Tail], S) :- sum(Tail,M), S is Head + M.
```
(**)

Order of computation in this rule is important ! We cannot do addition before the recursive call.
Can remove analysis, and keep only (*) and (**).

The predicate length(List,N)

The predicate *length*(*List*, *N*) is true if *N* is the number of elements in *List*.
Do case analysis, but explore a few simple cases before arriving at correct recursion.

If the input is the empty list, then its length is 0.

```
length([ ], 0).
```
(*)

If the input list has only one element *X*1, then its length is simply 1.

```
length([X1], 1).
```

If the input list has only 2 elements *X*1 and *X*2, then the length is 2.

```
length([X1,X2], L) :- L is 2.
```

If the input list has only 3 elements [*X*1, *X*2, *X*3], then to compute its length, add 1 to the length of [*X*1, *X*2].

```
length([X1,X2,X3], L) :- length([X1,X2],M), L is M+1.
```

For an arbitrary non-empty list, compute recursively the length of its tail, then add 1.

```
length([Head | Tail], L) :- length(Tail,M), L is M+1.
```
(**)

Order of computation in this rule is important ! We cannot do addition before the recursive call.
Can remove analysis, and keep only (*) and (**).

Lists as terms

Another example of terms. We can represent a list [*Head* | *Tail*] using a term *next*(*Head*, *Tail*). This somewhat resembles a linked list notation, but term notation is shorter. Also, we can represent the empty list as the constant *nil*.

Example: the usual list $[a, b, c] = [a \mid [b, c]] = [a \mid [b \mid c]]$ can be represented using terms as *next*(*a*, *next*(*b*, *next*(*c*, *nil*))).

Using this term-based representation of lists, we can write recursive programs over terms similarly to recursive programs over lists.

Exercise: re-implement the predicate *member*(*X*, *List*) using terms.

Accumulator technique

Example: *reverse*(*L*1, *L*2) holds if *L*2 is the list of elements from *L*1 in the opposite order. If we query $? -$ *reverse*([*a*, 1, 9, *b*], *L*2). then we expect that the list *L*2 = [*b*, 9, 1, *a*] will be returned as an output.

The predicate *reverse*(*L*1, *L*2) can be easily implemented using *append*. How? This is an exercise for you. But the question remains if it can be implemented without using *append*.

We can use an accumulator as an extra argument in a helping predicate.

reverse(*L*1, *L*2) $:-$ *revAux*(*L*1, [ ], *L*2).
revAux([ ], *L*, *L*).
revAux([*H*|*L*1], *L*2, *L*3) $:-$ *revAux*(*L*1, [*H*|*L*2], *L*3).

$? -$ *reverse*([*a*, *b*, *c*], *L*).
|
revAux([*a* | [*b*, *c*]], [ ], *L*)
|
revAux([*b* | [*c*]], [*a*], *L*).
|
revAux([*c* | [ ]], [*b*, *a*], *L*).
|
revAux([ ], [*c*, *b*, *a*], *L*).
|
success *L* = [*c*, *b*, *a*]

CPS721, Sections 051 to 091
Lecture 9

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Terms**
- **Constraints Satisfaction Problems (CSPs)**

Example: Families KB

To illustrate terms we consider a database with structured information about people and families. We introduce the following.

The predicate *family*(*Husband*, *Wife*, *ListOfChildren*) is true if the 1st and the 2nd argument represent a father and a mother of children in the 3rd argument

Two terms:

person(*FirstName*, *LastName*, *DOB*, *Job*)

date(*Day*, *Month*, *Year*)

These terms can be used as arguments of the predicate *family* to represent information in a more structured way.

Since *person* and *date* are terms, they can only occur inside the predicate *family* or as arguments of “equality" ($=$).

Example: consider the predicate “less than" and a term composed from the function “sum", both written with the usual notation, e.g.,

$$(a + b) + c \leq (b + c)$$

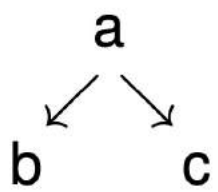
Notice terms are arguments of the predicate “$\leq$". But an arithmetical expression $(b + c)$ by itself would not have a truth value.

Binary Trees as Terms

We can use term with 3 arguments

```
tree(Element,LeftBranch,RightBranch)
```

to represent a binary tree. The empty tree can be represented by the constant `void` We represent the tree



as `tree(a, tree(b, void, void), tree(c,void,void))`

It is important to realize that binary tree is a recursive data structure: *a sub-tree of a tree is a tree itself*.

If we consider the root of the tree, then its left child is a root of a left sub-tree, which is also a tree. Similarly, its right child is a root of a right sub-tree, which is also a binary tree.

Consider a few recursive programs take advantage of recursiveness of binary trees.

As usual, we do case analysis to write recursive programs.

Recursion over terms

The predicate *binaryTree*(*X*) is true if *X* is a binary tree.
What is the base case ? What is a recursive case ?

```
binaryTree(void).

binaryTree( tree(Element, Left, Right) ) :-
          binaryTree(Left), binaryTree(Right).
```

The predicate *memberOfTree*(*Element*, *T*) is true if *Element* is a member of a binary tree *T*. To implement this predicate do case analysis.

```
/* memberOfTree(Element, tree(Root,Left,Right) ) :- Element=Root.*/
memberOfTree(Element, tree(Element,Left,Right) ).

memberOfTree(Element, tree(Node,Left,Right) ) :-
          memberOfTree(Element,Left).

memberOfTree(Element, tree(Node,Left,Right) ) :-
          memberOfTree(Element,Right).
```

Lists as terms

Another example of terms. We can represent a list [*Head* | *Tail*] using a term *next*(*Head*, *Tail*). This somewhat resembles a linked list notation, but term notation is shorter. Also, we can represent the empty list as the constant *nil*.

Example: the usual list $[a, b, c] = [a \mid [b, c]] = [a \mid [b \mid c]]$ can be represented using terms as $next(a,\ next(b,\ next(c,\ nil)))$.

Using this term-based representation of lists, we can write recursive programs over terms similarly to recursive programs over lists.

Exercise: re-implement the predicate *member*(*X*, *List*) using terms.

Accumulator technique

Example: *reverse*(*L*1, *L*2) holds if *L*2 is the list of elements from *L*1 in the opposite order. If we query $? -$ *reverse*([*a*, 1, 9, *b*], *L*2). then we expect that the list *L*2 = [*b*, 9, 1, *a*] will be returned as an output.

The predicate *reverse*(*L*1, *L*2) can be easily implemented using *append*. How? This is an exercise for you. But the question remains if it can be implemented without using *append*.

We can use an accumulator as an extra argument in a helping predicate.

reverse(*L*1, *L*2) $:-$ *revAux*(*L*1, [ ], *L*2).
revAux([ ], *L*, *L*).
revAux([*H*|*L*1], *L*2, *L*3) $:-$ *revAux*(*L*1, [*H*|*L*2], *L*3).

$? -$ *reverse*([*a*, *b*, *c*], *L*).
|
revAux([*a* | [*b*, *c*]], [ ], *L*)
|
revAux([*b* | [*c*]], [*a*], *L*).
|
revAux([*c* | [ ]], [*b*, *a*], *L*).
|
revAux([ ], [*c*, *b*, *a*], *L*).
|
success *L* = [*c*, *b*, *a*]

Agenda

- **Terms**
- **Constraints Satisfaction Problems (CSPs)**

Constraint satisfaction

Many challenging tasks that appear to require thinking have this form:

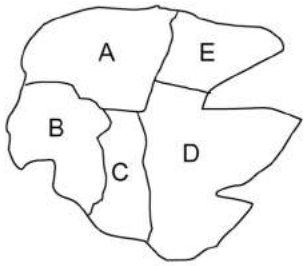
- some choices to be made
- some constraints to be satisfied

In this section, we will examine a collection of such reasoning tasks:

1. map colouring
2. Sudoku
3. cryptarithmetic
4. 8 queens
5. logic puzzles
6. scheduling

all of which can be solved in Prolog in a similar way.

Colouring a map



Five countries: `A, B, C, D, E`

Three colours: `red, white, blue`

Constraint: Neighbouring countries must have different colours on the map.

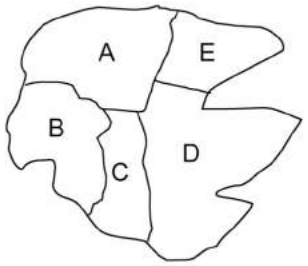
/* There are $3 * 3 * 3 * 3 * 3 = 3^5$ combinations, but not all are valid.*/

```
% solution(A,B,C,D,E) holds if A,B,C,D,E are colors
% that solve a map-coloring problem from the text.
solution(A,B,C,D,E) :-
   color(A), color(B), color(C), color(D), color(E),
   \+ A=B, \+ A=C, \+ A=D, \+ A=E, \+ B=C, \+ C=D, \+ D=E.

% The three colors are these.
color(red).
color(white).
color(blue).
```

Then we get:

```
?- solution(A,B,C,D,E).
A=red  B=white  C=blue  D=white  E=blue    % + other solutions too
```

Colouring a map using list as input



Five countries: `A, B, C, D, E`

Three colours: `red, white, blue`

Constraint: Neighbouring countries must have different colours on the map.

/* There are $3 * 3 * 3 * 3 * 3 = 3^5$ combinations, but not all are valid.*/

```
colour(blue).
colour(white).
colour(red).

solve([A,B,C,D,E]) :-                          /* Use predicate  solve(List) */

      colour(A), colour(B), colour(C),          /* Generate values */
      colour(D), colour(E),                     /* for the variables */

      not A=B, not A=C, not A=D, not A=E,       /* test the values */
      not B=C, not C=D, not D=E.                /* with constraints */.
```

Constraint satisfaction problems

The general form of these problems is this:

- some number of *variables*

  e.g. colours on the map

- values to be chosen from finite *domains*

  e.g. each colour must be one of red, white or blue

- there are *constraints* among subsets of the variables

  e.g. adjacent countries have different colours

A *solution* to a constraint satisfaction problem is an assignment of a value to each variable (taken from the domain of the variable) such that all of the constraints are satisfied.

Using generate and test

The simplest way to solve a constraint satisfaction problem using Prolog is to use generate and test.

So the general form is like this (see the map colouring):

```
solution(Variable_1, ..., Variable_n) :-
    domain_1(Variable_1),
        ...
    domain_n(Variable_n),
    constraint_1(Variable, ..., Variable),
        ...
    constraint_m(Variable, ..., Variable).
```

Sometimes the generation is interleaved with the testing.

Using generate and test

The simplest way to solve a constraint satisfaction problem using Prolog is to use generate and test.

So the general form is like this (see the map colouring):

```
solution(Variable_1, ..., Variable_n) :-
    domain_1(Variable_1),
        ...
    domain_n(Variable_n),
    constraint_1(Variable, ..., Variable),
        ...
    constraint_m(Variable, ..., Variable).
```

Sometimes the generation is interleaved with the testing.

Output in Prolog

It is often convenient to be able to produce output in Prolog other than just the values of variables.

Prolog provides two special atoms for queries or bodies of clauses:

- `write(`*term*`)`

  always succeeds and has the effect of printing the term

- `nl`

  always succeeds and has the effect of starting a new line

```
?- X=blue, nl, write('The value of X is '), write(X),
   write(', I believe,'), nl, write('  and that is it!').
The value of X is blue, I believe,
  and that is it!
X = blue
Yes
```

Output for map colouring

For example, for the map colouring, imagine we have the previous program together with the following additional clause:

```
print_colours :-
  solution(A,B,C,D,E), nl,
  write('Country A is coloured '), write(A), nl,
  write('Country B is coloured '), write(B), nl,
  write('Country C is coloured '), write(C), nl,
  write('Country D is coloured '), write(D), nl,
  write('Country E is coloured '), write(E), nl.
```

Then we get:

```
?- print_colours.      % Note that the query now has no variables

Country A is coloured red
Country B is coloured white
Country C is coloured blue
Country D is coloured white
Country E is coloured blue

Yes
```

Using generate and test (Summary)

- some number of variables
  – e.g. countries on the map that can be assigned colours
- values to be chosen from finite domains
  – e.g. each colour must be red, white, or blue
- constraints among subsets of variables
  – e.g. adjacent countries have different colours

The simplest program is "generate and test" (in pseudo-code)

```
solve(ListOfVariables) :-
      domain1 (Variable1 ),                          /*generate values */
              ...
      domainn (Variablen ),                          /*for the variables*/
      constraint1 (Variable1, ..., Variablen),       /*test the values*/
              ...
      constraintm (Variable1, ..., Variablen).       /*with constraints*/
```

and optionally

```
print_solution(ListOfVariables) :-
        solve(ListOfVariables),
        write("The answer is "), ..., nl.
```

The main conceptual challenge when solving problems with constraints: what input should be encoded as variables, and what the domains for variables can be?

Mapping to a graph

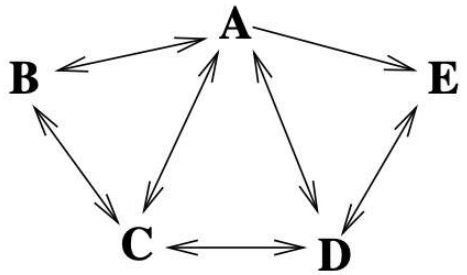
Consider the task of colouring the states on the map of continental USA. There are 50 states, and the task can be solved with 4 colours. What is the total number of combinations ?

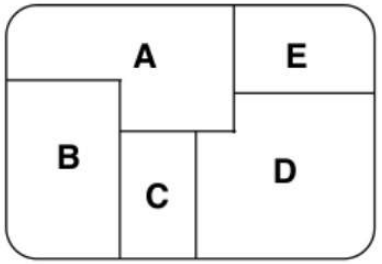
There are 4^{50} combinations. But neighbours must have *different* colours on the map. Lots of search is required, no matter what programming language you use to write a program.

How can we approach the task of map colouring?

Solution: Find all legal combinations of colours with minimal back-tracking.
Use a new tool: the dependency graph.

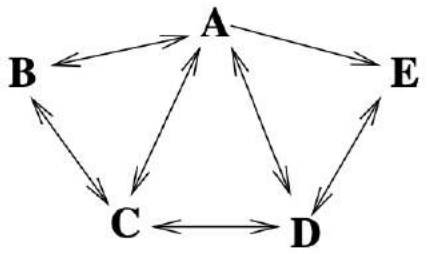
There is an edge from country *X* to *Y*, if the colour of *X* depends on the colour of *Y*.





Based on the dependency graph, we can design a more efficient algorithm.
Then, we can write a better program.

A (greedy) algorithm in pseudo-code



1. Guess an arbitrary colour of a country *A*.
2. Determine the colour of *B* such that it is different from the colour of *A*
3. Determine the colour of *C* from the constraints on *A*, *B*, *C*.
4. Determine the colour of *D* from the constraints on *A*, *C*, *D*.
5. Determine the colour of *E* from the constraints on *A*, *D*, *E*.

Now, we are prepared to write a more efficient program.

Interleave of generate and test

1. Guess an arbitrary colour of a country *A*.
2. Determine the colour of *B* such that it is different from the colour of *A*
3. Determine the colour of *C* from the constraints on *A*, *B*, *C*.
4. Determine the colour of *D* from the constraints on *A*, *C*, *D*.
5. Determine the colour of *E* from the constraints on *A*, *D*, *E*.

This is a new program design technique: *interleaving of generate and test*.

```
colour(blue).  colour(white).  colour(red).
solve([A,B,C,D,E]) :-                          /* Use predicate  solve(List) */
      colour(A), colour(B), not A=B,                  /* Steps (1) and (2) */
      colour(C), not C=B, not C=A,                    /* Step (3) */
      colour(D), not C=D, not D=A,                    /* Step (4) */
      colour(E), not E=D, not E=A.                    /* Step (5) */

print_solution([A,B,C,D,E]) :- solve( [A,B,C,D,E] ),
      write("The colour of A is "), write(A), nl,
      write("The colour of B is "), write(B), nl,
      write("The colour of C is "), write(C), nl,
      write("The colour of D is "), write(D), nl,
      write("The colour of E is "), write(E).
```

Arithmetic in Prolog

Prolog provides facilities for evaluating and comparing arithmetic expressions, made up of variables, numbers, parentheses, and the arithmetic operators $+$, $-$, $//$, $*$, $\hat{}$, and *mod*. For example

```
12*(X^2 -5) + (-6*Y//5) mod 4
```

Note that any variables appearing in such expressions must already have values.

Using expressions like these, Prolog has the following arithmetic predicates

$expr_1 > expr_2$, $expr_1 < expr_2$, $expr_1 >= expr_2$,
$expr_1 =< expr_2$, $expr_1 =:= expr_2$

We also have the statement

Var is *ArithmExpr*

If *Var* already has a value, the statement behaves just like *Var* **=:=** *ArithmExpr*

But if *Var* does not yet have a value, the statement succeeds, and *Var* gets the value of *ArithmExpr*.

Now, we are prepared to develop a first program.

Cryptarithmetic Puzzles

Cryptarithmetic puzzles are puzzles of the form

```
  SEND
+ MORE     % Each letter stands for a distinct digit
------     % Leading digits must not be 0
 MONEY
```

Variables: S, E, N, *etc.* and "carry digits"

Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for digits and $\{0, 1\}$ for carry digits

Constraints: unique digits, $S > 0$, $M > 0$, and

$$(D + E) \bmod 10 = Y \qquad (D + E)/10 = C_1$$

$$(N + R + C_1) \bmod 10 = E \qquad (N + R + C_1)/10 = C_{10}$$

$$(E + O + C_{10}) \bmod 10 = N \qquad (E + O + C_{10})/10 = C_{100}$$

$$(S + M + C_{100}) \bmod 10 = O \qquad (S + M + C_{100})/10 = M$$

A first program for SEND+MORE=MONEY

```
% solution(...) holds for a solution to SEND+MORE=MONEY.
solution(S,E,N,D,M,O,R,Y) :-
   uniq_digits(S,E,N,D,M,O,R,Y), S > 0, M > 0,
   Y is (D+E) mod 10, C1 is (D+E) // 10,
   E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
   N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10,
   O is (S+M+C100) mod 10, M is (S+M+C100) // 10.

% uniq(...) holds if the arguments are all distinct digits.
uniq_digits(S,E,N,D,M,O,R,Y) :-
   dig(S), dig(E), dig(N), dig(D), dig(M), dig(O), dig(R), dig(Y),
   \+ S=E, \+ S=N, \+ S=D, \+ S=M, \+ S=O, \+ S=R, \+ S=Y,
           \+ E=N, \+ E=D, \+ E=M, \+ E=O, \+ E=R, \+ E=Y,
                   \+ N=D, \+ N=M, \+ N=O, \+ N=R, \+ N=Y,
                           \+ D=M, \+ D=O, \+ D=R, \+ D=Y,
                                   \+ M=O, \+ M=R, \+ M=Y,
                                           \+ O=R, \+ O=Y,
                                                   \+ R=Y.

% The digits
dig(0). dig(1). dig(2). dig(3). dig(4).
dig(5). dig(6). dig(7). dig(8). dig(9).
```

A second program for SEND+MORE=MONEY

```
% solution(...) holds for a solution to SEND+MORE=MONEY.
solution([S,E,N,D,M,O,R,Y]) :-
        dig(S), dig(E), dig(N), dig(D), dig(M), dig(O), dig(R), dig(Y), % Generate
        uniq_digits([S,E,N,D,M,O,R,Y]). S > 0, M > 0,
        Y is (D+E) mod 10, C1 is (D+E) // 10,
        E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
        N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10, O is (S+M+C100) mod 10, M is
        (S+M+C100) // 10.

% uniq(...) holds if the arguments are all distinct digits.
uniq_digits([ ]).
uniq_digits([H | T]) :- not member(H, T), uniq_digits(T).


dig(0). dig(1). dig(2). dig(3). dig(4). dig(5). dig(6). dig(7). dig(8). dig(9).
```

Running this program

This programs works correctly:

```
?- solution(S,E,N,D,M,O,R,Y).    % SEND + MORE = MONEY
S = 9,                           % 9567 + 1085 = 10652
E = 5,
N = 6,
D = 7,
M = 1,
O = 0,
R = 8,
Y = 2
```

However there is a problem:

It takes over 90 seconds to find the solution, even on a very fast computer. (Try it!)

Can we do better?

We must try to minimize the guessing.

Minimize guessing: rule 1

Rule 1: Avoid guessing any value that is fully determined by other values and later testing if the guess is correct.

for example, instead of

```
uniq3(A,B,C),          % guess at A, B, and C
B is (A+C) mod 10      % then test if B is ok
```

we should use something like

```
uniq2(A,C),            % guess at A and C only
B is (A+C) mod 10      % calculate B once
uniq3(A,B,C)           % ensure they are all unique
```

The first version has a search space of $10 \times 10 \times 10 = 1000$.

The second has a search space of $10 \times 10 = 100$.

Minimize guessing: rule 2

Rule 2: Avoid placing independent guesses between the generation and testing of other values.

for example, instead of

```
dig(A), dig(B),     % guess at A and B
dig(C), dig(D),     % guess at C and D
A > B               % then test if A > B
```

we should use

```
dig(A), dig(B),     % guess at A and B
A > B,              % test if A > B
dig(C), dig(D),     % guess at C and D
```

In the first version, if we guess badly for A and B, we only get to reconsider *after* we have gone through *all* the values for C and D.

In the second version, if we guess badly for A and B, we reconsider immediately, before we consider any values for C and D.

Reordering the constraints

Here is a second version of the `solution` predicate.

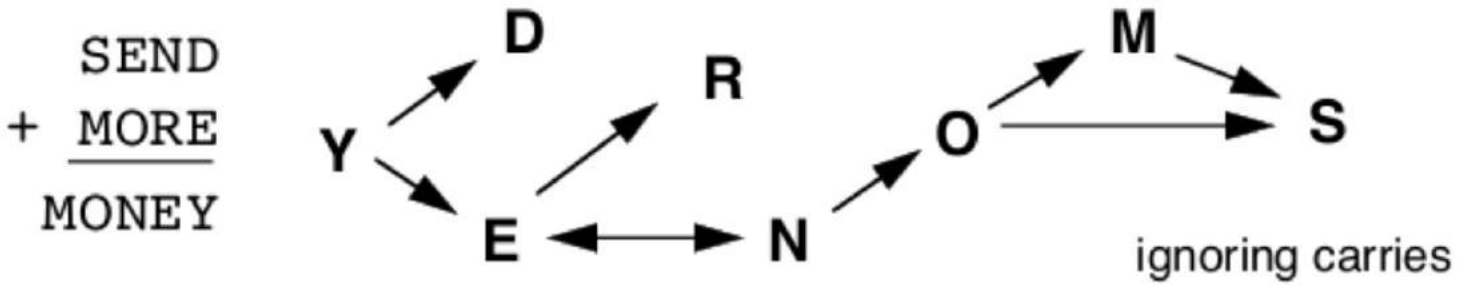
It uses exactly the same constraints, but in a *different order*.

```
% solution(...) holds for a solution to SEND+MORE=MONEY.
solution([S,E,N,D,M,O,R,Y]) :-
        dig(D), dig(E),
        Y is (D+E) mod 10, C1 is (D+E) // 10,
        dig(N), dig(R),
        E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
        dig(E), dig(O),
        N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10,
        dig(S), S > 0,  dig(M), M > 0,
        O is (S+M+C100) mod 10, M is (S+M+C100) // 10,
        uniq_digits([S,E,N,D,M,O,R,Y]).
% The rest of the program is as before.
```

It gives the same answer as before, but this time in .07 seconds!

Can we do even better?

Dependency graph



Draw a graph whose nodes are the variables, with an edge from Var_1 to Var_2 if the value of Var_1 will be calculated from that of Var_2.

Previous method was proceeding from the right to the left. It guessed at *D* and *E* first, then calculated *Y*, guessed at *N* and *R*, then verified *E*, guessed at *O*, verified *N*, etc.

Can use the dependency graph to reduce guessing:

- guess at *M* and *S*, calculate *O*
- guess at *E*, calculate *N*
- guess at *R*, verify choice for *E*
- guess at *D*, calculate *Y*

With this order, we will also need to guess carry digits.

Now, we can write a really efficient program.

Very efficient version

```
print_solution( [S,E,N,D,M,O,R,Y] ) :-
        solve( [S,E,N,D,M,O,R,Y] ),
        write(' '), write(S), write(E), write(N), write(D),nl,
        write('+ '), write(M), write(O), write(R), write(E),nl,
        write(' '),write("_____"),nl,
        write(' '),write(M),write(O),write(N),write(E),write(Y),nl.

car(0). car(1).
dig(0). dig(1). dig(2). dig(3). dig(4). dig(5). dig(6). dig(7). dig(8). dig(9).

solve([S,E,N,D,M,O,R,Y]) :-
    car(M), M > 0, % M can only be 1
    car(C100),
    dig(S), S > 0,
    M is (S+M+C100) // 10, O is (S+M+C100) mod 10,
    dig(E), car(C10),
    N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10,
    dig(R), car(C1),
    E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
    dig(D),
    Y is (D+E) mod 10, C1 is (D+E) // 10,
    uniq_digits([S,E,N,D,M,O,R,Y]).
```

CPS721, Sections 051 to 091
Lecture 10

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Constraint Satisfaction Problems (CSPs)**

Example: Scheduling

In general: finding times and/or places for events. Example:

We want to schedule a new class to meet 5 hours a week.
Meetings times are on the hour from 9am to 4pm, on weekdays.
Some of the times will be taken by already-scheduled courses.
Classes must **not** meet on successive hours on the same day,
and for more than 2 hours total on any one day.
Variables - ? Domain - ?

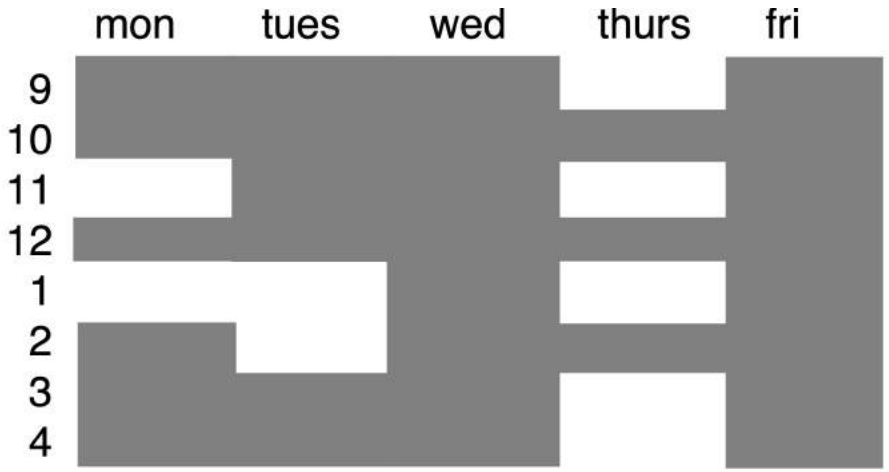
Variables: $T1$, $T2$, $T3$, $T4$, $T5$, where each T_i is one of the 5 meeting times.

Domain: pair [*Day*, *Hour*] where
Day is one of `mon, tues, wed, thurs, fri`
Hour is one of `9, 10,11, 12, 1, 2, 3, 4`

Constraints: the five chosen times must be
- non-consecutive
- no more than 2 per day /* anything else ? */
- not already taken
- all different. /* not explicitly mentioned */

Constraint: taken times

Suppose the following times are taken




```
taken([mon,9]).    taken([tues,9]).    taken([thurs,10]).
taken([mon,10]).   taken([tues,10]).   taken([thurs,2]).
taken([mon,2]).    taken([tues,11]).
taken([mon,3]).    taken([tues,3]).
taken([mon,4]).    taken([tues,4]).

taken([_,12]).
taken([wed,_]).
taken([fri,_]).
```

Note a possible solution can be:

```
TimeList = [[mon,11],[mon,1], [tues,1], [thurs,9],[thurs,11]]
```

Other constraints

Sometimes, it is more convenient to implement negations of constraints:

- instead of non-consecutive –> when two meetings are consecutive
- instead of "no more than 2 per day" –> meet 3 times per day

```
two_consecutive_hours(TimeList) :-
      member([Day,Hour1], TimeList),
      member([Day,Hour2], TimeList),
      append(X, [Hour1,Hour2| HH], [9,10,11,12,1,2,3,4]).

three_classes_same_day(TL) :-
      member([Day,Hour1], TL),
      member([Day,Hour2], TL),
      member([Day,Hour3], TL),
      not Hour1=Hour2, not Hour1=Hour3, not Hour2=Hour3.

append([],List,List).
append([X | L1],L2,[X | L3]) :- append(L1,L2,L3).

member(X, [X | List]).
member(X, [H | T]) :- member(X,Tail).
```

Scheduling in Prolog

Recall that the following atomic statements are given:

```
taken([mon,9]).
```
... and so on.

```
class_time( [Day,Hour] ) :-                    /* class time is a pair.*/
      member(Day, [mon,tues,wed,thurs,fri]),
      member(Hour, [9,10,11,12,1,2,3,4]).

solve( [T1,T2,T3,T4,T5] ) :-
      class_time(T1), class_time(T2), class_time(T3),
           class_time(T4), class_time(T5),
      not taken(T1), not taken(T2), not taken(T3),
           not taken(T4), not taken(T5),
      not two_consecutive_hours( [T1,T2,T3,T4,T5] ),
      not three_classes_same_day( [T1,T2,T3,T4,T5] ),
      all_diff( [T1,T2,T3,T4,T5] ).

all_diff([]).
all_diff([N | L]) :- not member(N,L), all_diff(L).
```

What design technique are we using in this program ?

How to improve the computational efficiency of this program ?

Logic puzzles

Example:

In conversation, Chris, Sandy and Pat discovered that they had distinct occupations and played distinct musical instruments. Also

1. *Chris is married to the doctor.*
2. *The lawyer plays the piano.*
3. *Chris is not the engineer.*
4. *Sandy is a patient of the violinist.*

Who plays the flute?

To solve puzzles like these, we need to determine what the variables are, what values they can have, and how the given information leads to constraints on possible solutions.

Logic puzzle as constraint satisfaction

To determine who plays the flute, it is clear that we will need to figure out who plays what, and who has what job.

This suggests the following:

Variables: `Doctor`, `Lawyer`, `Engineer`, `Piano`, `Violin`, `Flute`

Domain: {`sandy`, `chris`, `pat`}

For the constraints, we need to use the fact that

- if x is married to y, then $x \neq y$.
- if x is a patient of y, then $x \neq y$ and y is the doctor.

Solving the puzzle in Prolog for Flute

```
% A logic puzzle involving jobs and musical instruments,
solution(Flute) :-

   % Distinct occupations and instruments
   uniq_people(Doctor,Lawyer,Engineer),
   uniq_people(Piano,Violin,Flute),

   % The four clues
   \+ chris = Doctor,     % Chris is married to the doctor.
   Lawyer = Piano,        % The lawyer plays the piano.
   \+ Engineer = chris,   % The engineer is not Chris.
   Violin = Doctor,       % Sandy is a patient of
   \+ sandy = Violin.     %    the violinist.

% uniq(...) is used to generate three distinct people.
uniq_people(A,B,C) :- person(A),  person(B),  person(C),
                      \+ A=B, \+ A=C, \+ B=C.

% The three given people
person(chris).   person(sandy).   person(pat).
```

Solving the puzzle in Prolog for everyone

```
/* Generating the people + testing for distinct */
person(chris).  person(sandy).  person(pat).
distinct_people(X,Y,Z) :-
        person(X), person(Y), person(Z),
        not X=Y, not X=Z, not Y=Z.

solve( [Doctor,Lawyer,Engineer,Piano,Violin,Flute] ) :-
    distinct_people(Doctor,Lawyer,Engineer),
    distinct_people(Piano,Violin,Flute),

    /* Chris is married to the doctor */
    not chris = Doctor,

    /* The lawyer plays the piano */
    Lawyer = Piano,

    /* The engineer is not Chris */
    not Engineer = chris,

    /* Sandy is a patient of the violinist */
    Violin = Doctor,
     not sandy = Violin.
```

Hidden Variables

Sometimes the only way to express the constraints is to imagine that there are variables other than the ones mentioned in the puzzle.

cf. the carry digits in cryptarithmetic

Example: as before with Chris, Sandy, and Pat, except that (3) is

3. *Pat is not married to the engineer.*

It is useful to think of terms of new variables, for Chris' spouse, Sandy's spouse, and Pat's spouse.

- Domain: Chris, Sandy, Pat or *none*
- Constraints: only 4 legal arrangements
  - — nobody is married
  - — Chris and Sandy
  - — Chris and Pat
  - — Sandy and Pat

The logic puzzle revisited

```
% A second logic puzzle involving jobs and musical instruments
solution(Flute) :-

   uniq_people(Doctor,Lawyer,Engineer),
   uniq_people(Piano,Violin,Flute),

   % Generate values for the three spouse variables.
   spouses(Chris_spouse,Sandy_spouse,Pat_spouse),

   Chris_spouse = Doctor,      % Chris is married to the doctor.
   Lawyer = Piano,             % The lawyer plays the piano.
   \+ Pat_spouse = Engineer,   % Pat is not married to the engineer.
   Violin = Doctor,            % Sandy is a patient of
   \+ sandy = Violin.          %    the violinist.

uniq_people(A,B,C) :-
   person(A),  person(B),  person(C), \+ A=B, \+ A=C, \+ B=C.

person(chris).   person(sandy).   person(pat).

% spouses(X,Y,Z): X,Y,Z can be spouses of Chris,Sandy,Pat.
spouses(none,none,none).      % Nobody is married.
spouses(sandy,chris,none).    % Chris and Sandy are married.
spouses(pat,none,chris).      % Chris and Pat are married.
spouses(none,pat,sandy).      % Sandy and Pat are married.
```

Program with variables for spouses (general case)

```
person(chris).  person(sandy).  person(pat).
distinct_people(X,Y,Z) :-
        person(X), person(Y), person(Z),
        not X=Y, not X=Z, not Y=Z.

/* spouses(ChrisSpouse, SandySpouse, PatSpouse) */
spouses(none,none,none).                          /* nobody married */
spouses(sandy,chris,none).                        /* Chris and Sandy */
spouses(pat,none,chris).                          /* Chris and Pat */
spouses(none,pat,sandy).                          /* Sandy and Pat */

solve( [Doctor,Lawyer,Engineer,Piano,Violin,Flute] ) :-
    distinct_people(Doctor,Lawyer,Engineer),
    distinct_people(Piano,Violin,Flute),
   spouses(Chris_spouse,Sandy_spouse,Pat_spouse),

   Chris_spouse = Doctor,                         /* new formulation */
   Lawyer = Piano,
    not Pat_spouse = Engineer,                    /* new constraint */
   Violin = Doctor,
    not sandy = Violin.
```

Printing Solution

```
print_solution( [Doctor,Lawyer,Engineer,Piano,Violin,Flute] ) :-
  solve([Doctor,Lawyer,Engineer,Piano,Violin,Flute]),
  write('The doctor is '), write(Doctor), nl,
  write('The lawyer is '), write(Lawyer), nl,
  write('The engineer is '), write(Engineer), nl,
  write('================================='), nl,
  write('The piano is played by '), write(Piano), nl,
  write('The violin is played by '), write(Violin), nl,
  write('The flute is played by '), write(Flute), nl.
```

CPS721, Sections 051 to 091
Lecture 11

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Constraint Satisfaction Problems (CSPs)**

Constraint satisfaction problems

The general form of these problems is this:

- some number of *variables*

  e.g. colours on the map

- values to be chosen from finite *domains*

  e.g. each colour must be one of red, white or blue

- there are *constraints* among subsets of the variables

  e.g. adjacent countries have different colours

A *solution* to a constraint satisfaction problem is an assignment of a value to each variable (taken from the domain of the variable) such that all of the constraints are satisfied.

Cryptarithmetic Puzzles

Cryptarithmetic puzzles are puzzles of the form

```
  SEND
+ MORE     % Each letter stands for a distinct digit
------     % Leading digits must not be 0
 MONEY
```

Variables: S, E, N, *etc.* and "carry digits"

Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for digits and $\{0, 1\}$ for carry digits

Constraints: unique digits, $S > 0$, $M > 0$, and

$$(D + E) \bmod 10 = Y \qquad (D + E)/10 = C_1$$

$$(N + R + C_1) \bmod 10 = E \qquad (N + R + C_1)/10 = C_{10}$$

$$(E + O + C_{10}) \bmod 10 = N \qquad (E + O + C_{10})/10 = C_{100}$$

$$(S + M + C_{100}) \bmod 10 = O \qquad (S + M + C_{100})/10 = M$$

A second program for SEND+MORE=MONEY

```
% solution(...) holds for a solution to SEND+MORE=MONEY.
solution([S,E,N,D,M,O,R,Y]) :-
        dig(S), dig(E), dig(N), dig(D), dig(M), dig(O), dig(R), dig(Y), % Generate
        uniq_digits([S,E,N,D,M,O,R,Y]). S > 0, M > 0,
        Y is (D+E) mod 10, C1 is (D+E) // 10,
        E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
        N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10, O is (S+M+C100) mod 10, M is
        (S+M+C100) // 10.

% uniq(...) holds if the arguments are all distinct digits.
uniq_digits([ ]).
uniq_digits([H | T]) :- not member(H, T), uniq_digits(T).



dig(0). dig(1). dig(2). dig(3). dig(4). dig(5). dig(6). dig(7). dig(8). dig(9).
```

Takes 90 seconds!

Reordering the constraints

Here is a second version of the `solution` predicate.

It uses exactly the same constraints, but in a *different order*.

```
% solution(...) holds for a solution to SEND+MORE=MONEY.
solution([S,E,N,D,M,O,R,Y]) :-
        dig(D), dig(E),
        Y is (D+E) mod 10, C1 is (D+E) // 10,
        dig(N), dig(R),
        E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
        dig(E), dig(O),
        N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10,
        dig(S), S > 0,  dig(M), M > 0,
        O is (S+M+C100) mod 10, M is (S+M+C100) // 10,
        uniq_digits([S,E,N,D,M,O,R,Y]).
% The rest of the program is as before.
```

It gives the same answer as before, but this time in .07 seconds!

Can we do even better?

Very efficient version

```
print_solution( [S,E,N,D,M,O,R,Y] ) :-
        solve( [S,E,N,D,M,O,R,Y] ),
        write(' '), write(S), write(E), write(N), write(D),nl,
        write('+ '), write(M), write(O), write(R), write(E),nl,
        write(' '),write("_____"),nl,
        write(' '),write(M),write(O),write(N),write(E),write(Y),nl.

car(0). car(1).
dig(0). dig(1). dig(2). dig(3). dig(4). dig(5). dig(6). dig(7). dig(8). dig(9).

solve([S,E,N,D,M,O,R,Y]) :-
    car(M), M > 0, % M can only be 1
    car(C100),
    dig(S), S > 0,
    M is (S+M+C100) // 10, O is (S+M+C100) mod 10,
    dig(E), car(C10),
    N is (E+O+C10) mod 10, C100 is (E+O+C10) // 10,
    dig(R), car(C1),
    E is (N+R+C1) mod 10, C10 is (N+R+C1) // 10,
    dig(D),
    Y is (D+E) mod 10, C1 is (D+E) // 10,
    uniq_digits([S,E,N,D,M,O,R,Y]).
```

Logic puzzles

Example:

In conversation, Chris, Sandy and Pat discovered that they had distinct occupations and played distinct musical instruments. Also

1. *Chris is married to the doctor.*
2. *The lawyer plays the piano.*
3. *Chris is not the engineer.*
4. *Sandy is a patient of the violinist.*

Who plays the flute?

To solve puzzles like these, we need to determine what the variables are, what values they can have, and how the given information leads to constraints on possible solutions.

Logic puzzle as constraint satisfaction

To determine who plays the flute, it is clear that we will need to figure out who plays what, and who has what job.

This suggests the following:

Variables: `Doctor`, `Lawyer`, `Engineer`, `Piano`, `Violin`, `Flute`

Domain: {`sandy`, `chris`, `pat`}

For the constraints, we need to use the fact that

- if x is married to y, then $x \neq y$.
- if x is a patient of y, then $x \neq y$ and y is the doctor.

Solving the puzzle in Prolog for Flute

```
% A logic puzzle involving jobs and musical instruments,
solution(Flute) :-

   % Distinct occupations and instruments
   uniq_people(Doctor,Lawyer,Engineer),
   uniq_people(Piano,Violin,Flute),

   % The four clues
   \+ chris = Doctor,     % Chris is married to the doctor.
   Lawyer = Piano,        % The lawyer plays the piano.
   \+ Engineer = chris,   % The engineer is not Chris.
   Violin = Doctor,       % Sandy is a patient of
   \+ sandy = Violin.     %    the violinist.

% uniq(...) is used to generate three distinct people.
uniq_people(A,B,C) :- person(A),  person(B),  person(C),
                      \+ A=B, \+ A=C, \+ B=C.

% The three given people
person(chris).   person(sandy).   person(pat).
```

Solving the puzzle in Prolog for everyone

```
/* Generating the people + testing for distinct */
person(chris).  person(sandy).  person(pat).
distinct_people(X,Y,Z) :-
        person(X), person(Y), person(Z),
        not X=Y, not X=Z, not Y=Z.

solve( [Doctor,Lawyer,Engineer,Piano,Violin,Flute] ) :-
    distinct_people(Doctor,Lawyer,Engineer),
    distinct_people(Piano,Violin,Flute),

    /* Chris is married to the doctor */
    not chris = Doctor,

    /* The lawyer plays the piano */
    Lawyer = Piano,

    /* The engineer is not Chris */
    not Engineer = chris,

    /* Sandy is a patient of the violinist */
    Violin = Doctor,
     not sandy = Violin.
```

Hidden Variables

Sometimes the only way to express the constraints is to imagine that there are variables other than the ones mentioned in the puzzle.

cf. the carry digits in cryptarithmetic

Example: as before with Chris, Sandy, and Pat, except that (3) is

3. *Pat is not married to the engineer.*

It is useful to think of terms of new variables, for Chris' spouse, Sandy's spouse, and Pat's spouse.

- Domain: Chris, Sandy, Pat or *none*
- Constraints: only 4 legal arrangements
  - — nobody is married
  - — Chris and Sandy
  - — Chris and Pat
  - — Sandy and Pat

The logic puzzle revisited

```
% A second logic puzzle involving jobs and musical instruments
solution(Flute) :-

   uniq_people(Doctor,Lawyer,Engineer),
   uniq_people(Piano,Violin,Flute),

   % Generate values for the three spouse variables.
   spouses(Chris_spouse,Sandy_spouse,Pat_spouse),

   Chris_spouse = Doctor,      % Chris is married to the doctor.
   Lawyer = Piano,             % The lawyer plays the piano.
   \+ Pat_spouse = Engineer,   % Pat is not married to the engineer.
   Violin = Doctor,            % Sandy is a patient of
   \+ sandy = Violin.          %    the violinist.

uniq_people(A,B,C) :-
   person(A),  person(B),  person(C), \+ A=B, \+ A=C, \+ B=C.

person(chris).   person(sandy).   person(pat).

% spouses(X,Y,Z): X,Y,Z can be spouses of Chris,Sandy,Pat.
spouses(none,none,none).       % Nobody is married.
spouses(sandy,chris,none).     % Chris and Sandy are married.
spouses(pat,none,chris).       % Chris and Pat are married.
spouses(none,pat,sandy).       % Sandy and Pat are married.
```

Program with variables for spouses (general case)

```
person(chris).  person(sandy).  person(pat).
distinct_people(X,Y,Z) :-
        person(X), person(Y), person(Z),
        not X=Y, not X=Z, not Y=Z.

/* spouses(ChrisSpouse, SandySpouse, PatSpouse) */
spouses(none,none,none).                          /* nobody married */
spouses(sandy,chris,none).                        /* Chris and Sandy */
spouses(pat,none,chris).                          /* Chris and Pat */
spouses(none,pat,sandy).                          /* Sandy and Pat */

solve( [Doctor,Lawyer,Engineer,Piano,Violin,Flute] ) :-
    distinct_people(Doctor,Lawyer,Engineer),
    distinct_people(Piano,Violin,Flute),
   spouses(Chris_spouse,Sandy_spouse,Pat_spouse),

   Chris_spouse = Doctor,                         /* new formulation */
   Lawyer = Piano,
    not Pat_spouse = Engineer,                    /* new constraint */
   Violin = Doctor,
    not sandy = Violin.
```

Printing Solution

```
print_solution( [Doctor,Lawyer,Engineer,Piano,Violin,Flute] ) :-
  solve([Doctor,Lawyer,Engineer,Piano,Violin,Flute]),
  write('The doctor is '), write(Doctor), nl,
  write('The lawyer is '), write(Lawyer), nl,
  write('The engineer is '), write(Engineer), nl,
  write('================================='), nl,
  write('The piano is played by '), write(Piano), nl,
  write('The violin is played by '), write(Violin), nl,
  write('The flute is played by '), write(Flute), nl.
```

Wedding Puzzle

There are 5 women: Anne, Cathy, Eve, Fran, Ida. Also there are 5 men: Paul, Rob, Stan, Vern, Wally. Five couples were married last week, each on a different weekday. From the information provided, determine the woman and man who make up each couple, as well as the day on wich each couple was married.

1. Anne was married on Monday, but not with Wally.
2. Stan's wedding was on Wednesday.
3. Rob was married on Friday, but not to Ida.
4. Vern (who married Fran) was married the day after Eve.

Write a Prolog program using smart interleaving of generate and test to solve this CSP problem. It you use any helping predciates, you must implement them.

What are the unknown variables ? What are the domains for the variables ? We have to make sure the constraints are easy to represent.

Since each married couple is uniquely associated with a weekday, it is natural to encode the underlying domain with the atomic statements

```
day(1).  day(2).  day(3).  day(4).  day(5).
```

The variables are the first letters of the names of the women A, C, E, F, I and the men P, R, S, V, W. Each couple is mapped to same day. How to write a program to solve this puzzle ?

Wedding puzzle: solution

1. Anne was married on Monday, but not with Wally.
2. Stan's wedding was on Wednesday.
3. Rob was married on Friday, but not to Ida.
4. Vern (who married Fran) was married the day after Eve.

```
day(1).  day(2).  day(3).  day(4).  day(5).

solve([A,C,E,F,I,P,R,S,V,W]) :-
      day(A), A=1, day(W), not A=W,              /*(1) We still call day(A) */
      day(S), S=3,        /*(2) Call day(S) since we have to follow the technique*/
      day(R), R=5, day(I), not R=I,                                    /* 3 */
      day(V), day(F), V=F, day(E), V is E+1, /* (4) implicit constraints? */
      day(C),                                  /* implicit: Cathy got married */
      day(P),                                   /* implicit: Paul got married */
      all_diff([A,C,E,F,I]),          /* women got married on different days */
      all_diff([P,R,S,V,W]).            /* men got married on different days */

all_diff([ ]).
all_diff([X | List]) :- not member(X,List), all_diff(List).
```

Wages puzzle

Ann, *Bob*, *Cindy*, *Daniel*, *Eve* are working part-time at a company that has several positions with different pay levels. There are the following wages per week that the employees can earn depending on their position within the company: *$100, $200, $300, $400, $500*. Find wages of all people based on the following constraints.

1. *Bob*'s wage is different both from *Cindy*'s wage and from *Eve*'s wage.
2. *Cindy* earns less than *Daniel*, and less than *Eve*.
3. *Ann*'s wage is lower than *Eve*'s wage and it is lower than *Bob*'s wage.
4. *Ann*'s wage is different both from *Daniel*'s wage and from *Cindy*'s wage.
5. *Daniel* earns more than *Eve* and more than *Bob*.

Write an efficient Prolog program that computes all 5 possible wages subject to all of the given constraints. The program has to follow the interleaving of generate and test technique.

What will be the variables ? What are the domains for variables ? Make sure we can easily formulate the constraints.

Since the puzzle is formulated in terms of wages, it is natural to choose the wages $100, 200, 300, 400, 500$ as the domain.

```
wage(100).  wage(200).  wage(300).  wage(400).  wage(500).
```

How to start writing a program that solves this problem?

Wages puzzle

1. *Bob*'s wage is different both from *Cindy*'s wage and from *Eve*'s wage.
2. *Cindy* earns less than *Daniel*, and less than *Eve*.
3. *Ann*'s wage is lower than *Eve*'s wage and it is lower than *Bob*'s wage.
4. *Ann*'s wage is different both from *Daniel*'s wage and from *Cindy*'s wage.
5. *Daniel* earns more than *Eve* and more than *Bob*.

```
wage(100).  wage(200).  wage(300).  wage(400).  wage(500).

solve([A,B,C,D,E]) :-
      wage(B), wage(C), not B=C, wage(E), not B=E,          /* 1 */
      C < E,                                                /* 2 */
     wage(D), C < D,                                        /* 2 */
      wage(A), A < E, A < B,                                /* 3 */
     not A=D, not A=C,                                      /* 4 */
     D > E, D > B.                                          /* 5 */
```

Puzzle about a quiz

The following puzzle can be formulated as a constraint satisfaction problem (CSP). A group of 5 students wrote a quiz that included 5 questions with true/false answers. The answers submitted by these students are as follows

Ann :	*true*,	*true*,	*false*,	*true*,	*false*
Bob :	*false*,	*true*,	*true*,	*true*,	*false*
Cindy :	*true*,	*false*,	*true*,	*true*,	*false*
David :	*false*,	*true*,	*true*,	*false*,	*true*
Edward :	*true*,	*false*,	*true*,	*false*,	*true*

Once students have received their grades (calculated as the number of correct answers), they found that

1. Cindy got more answers right than Ann.
2. David got more right than Bob.
3. Edward did not get all the answers right, nor did he get them all wrong.

It turns out that these constraints are sufficient to determine the correct answers to all questions from the quiz! Write a Prolog program that can find the correct answers to all the questions. (**Do not waste your time on solving this puzzle yourself**: your Prolog program should solve it!)

Puzzle about a quiz

(1) (**0.5 point**) Using the single predicate **answer(X)** characterize the finite domain of 2 possible answers to each question: `t` and `f`.

(2) (**1.5 point**) Implement a helping predicate `grade(Q1,Q2,Q3,Q4,Q5,A1,A2,A3,A4,A5,N)` that holds when *N* is the grade (the number of correct answers) when a student submits the Q_i while the correct answers are the A_i. In other words, the predicate `grade` takes as inputs Q_i, A_i, $i \in \{1, 2, 3, 4, 5\}$, counts the number *N* of matches between the corresponding elements Q_i and A_i, and returns *N* as an output. *Hint:* it is easy to implement this predicate using another auxiliary predicate `sameAs(L1,L2,N)` that counts the number of total matches between the corresponding elements of lists `L1` and `L2`, where `L1` and `L2` have equal length. For example,
`sameAs([t,f,f,f,t],[t,t,t,t,t],N)` returns `N=2`, but
`sameAs([t,f,t,f,t],[f,t,f,t,f],N)` returns `N=0`.

(3) (**3 points**) Write a Prolog program that solves this instance of CSP. You can use any of the techniques that we studied in class: *generate-and-test* or *interleaving of generate and test*. Provide in-line comments indicating which constraints you implement where. Use the predicates `answer` and `grade` in your program. *You do* **not** *need any other helping predicates in your program, but you decide to use them, then implement them as well.*

Helping predicates

As usual, do case analysis: what is the base case(s), then recursive rule(s).

```
same([A],[A],1).                                    /* Base cases */
same([S],[A],0) :- not A=S.
```

Case 1: if the heads of L1 and L2 match, then add 1 to the counter.
Case 2: they do not match, do not change the counter.

```
sameAs([H | List1], [H | List2], N) :-                    /* Case 1 */
      sameAs(List1, List2, S), N is S+1.

sameAs([H1 | Tail1], [H2 | Tail2], N) :- not H1=H2,    /*Case 2*/
      sameAs(Tail1, Tail2, N).
```

How to implement `grade(Q1,Q2,Q3,Q4,Q5,A1,A2,A3,A4,A5,N)` ?

```
grade(Q1,Q2,Q3,Q4,Q5, A1,A2,A3,A4,A5, N) :
       sameAs([Q1,Q2,Q3,Q4,Q5], [A1,A2,A3,A4,A5], N).
```

Use the predicates `answer` and `grade` in your program.

Solution to puzzle about a quiz

Ann :	*true*,	*true*,	*false*,	*true*,	*false*
Bob :	*false*,	*true*,	*true*,	*true*,	*false*
Cindy :	*true*,	*false*,	*true*,	*true*,	*false*
David :	*false*,	*true*,	*true*,	*false*,	*true*
Edward :	*true*,	*false*,	*true*,	*false*,	*true*

Once students have received their grades (calculated as the number of correct answers), they found that

1. Cindy got more answers right than Ann.
2. David got more right than Bob.
3. Edward did not get all the answers right, nor did he get them all wrong.

```
answer(t).  answer(f).
solve([A1,A2,A3,A4,A5]) :-   /*find correct quiz answers only */
      answer(A1), answer(A2),                    /* generate values for variables */
     answer(A3), answer(A4), answer(A5),
     grade(t,t,f,t,f,A1,A2,A3,A4,A5,NA),     /* NA is the grade for Ann */
     grade(f,t,t,t,f,A1,A2,A3,A4,A5,NB),     /* NB is the grade for Bob */
     grade(t,f,t,t,f,A1,A2,A3,A4,A5,NC),    /*NC is the grade for Cindy */
     grade(f,t,t,f,t,A1,A2,A3,A4,A5,ND),          /*grade for David */
     grade(t,f,t,f,t,A1,A2,A3,A4,A5,NE),          /*grade for Edward */
     NC > NA,                                      /* the 1st constraint */
     ND > NB,                                      /* the 2nd constraint */
     NE > 0, NE < 5.                          /* how to improve this program ? */
```

CPS721, Sections 051 to 091
Lecture 12

Instructor: Nariman Farsad

*** The slides in this deck are adopted from Hector Levesque's textbook and accompanying slides and slides by Mikhail Soutchanski.**

Agenda

- **Exam Review**

Queries to a KB

Assume that you are given a collection of atomic statements about products available for purchase, manufacturers and prices, using the following predicates:

inStore(ItemID,ProductType,Quantity) – *Quantity* of a *ProductType* (it can be a phone, a laptop, a tablet, etc) with an unique *ItemID* (it can be any unique item identifier) is available in the store, where *Quantity* is the number of the copies of a product that are available in the store.

manufacturer(ItemID,Company,Year) – *ItemID* has been manufactured by *Company* in *Year*.

Formulate the following query in Prolog using these predicates: Find the most popular product available in a store, i.e., the product with the largest quantity.

```
?- inStore(ItemID,Product,Q),
        not ( inStore(ItemID2,Product2,Q2),  Q2 > Q  )
```

Find the Apple manufactured product with the largest quantity available in a store.

```
?- inStore(ID,P,Q), manufacturer(ID,apple,Year),
        not (  inStore(ID2,P2,Q2),  manufacturer(ID2,apple,Y2),  Q2 > Q)
```

Notice the standard pattern in queries of this kind. Questions ?

Matching lists

Find if the following lists match or not. Provide transformations.

`[K, L | [M | N] ]` and `[ cps | [mth | [phy]] ]`

`[K, L | [ M | N ] ] = [K, L, M | N]`

`[ cps | [mth | [phy] ] ] = [ cps | [mth, phy] ]`

`[ cps | [ mth, phy] ] = [ cps, mth, phy]`

`[K, L, M | N ] =`
`[cps,mth,phy | [ ] ]` if `K=cps, L=mth, M=phy, N=[ ]`

Can the lists `[X, c, b | [c | X ] ]` and `[[p,q] | [c,b,c | [p,q,r] ]]` match or not?

`[X, c, b | [c | X ] ] = [X, c, b, c | X ]`

`[[p,q] | [c,b,c | [p,q,r] ] ] = [[p,q], c,b,c | [p,q,r] ]`

`[ X, c, b, c | X ] =?=`
`[[p,q], c,b,c | [p,q,r] ]` if `X=[p,q]` and `X=[p,q,r]` at the same time, but this is impossible. So, the given lists do not match.

Typical errors in recursion

The predicate *length*(*List*, *N*) is true if *N* is the number of elements in *List*.
Recall what is the recursive program that implements this predicate ?

```
length([ ],0).
length([ H | T ], N) :- length(T, Interm), N is Interm + 1.
```

There are several typical errors that students make when writing a recursive rule in the programs of this type.

```
length([ H | T ], N) :- N is X + 1, length(T, X).
```

Error: the output X must be computed first from a recursive call, before it an be used to compute the number of elements N of the given list.

```
length([ H | T ], N) :- length(T, X), N = X + 1.
```

Error: to compute addition of X and 1, we must use **is** since equality "=" only unifies strings on the left and on the right sides, but it does not compute a sum of X and 1.

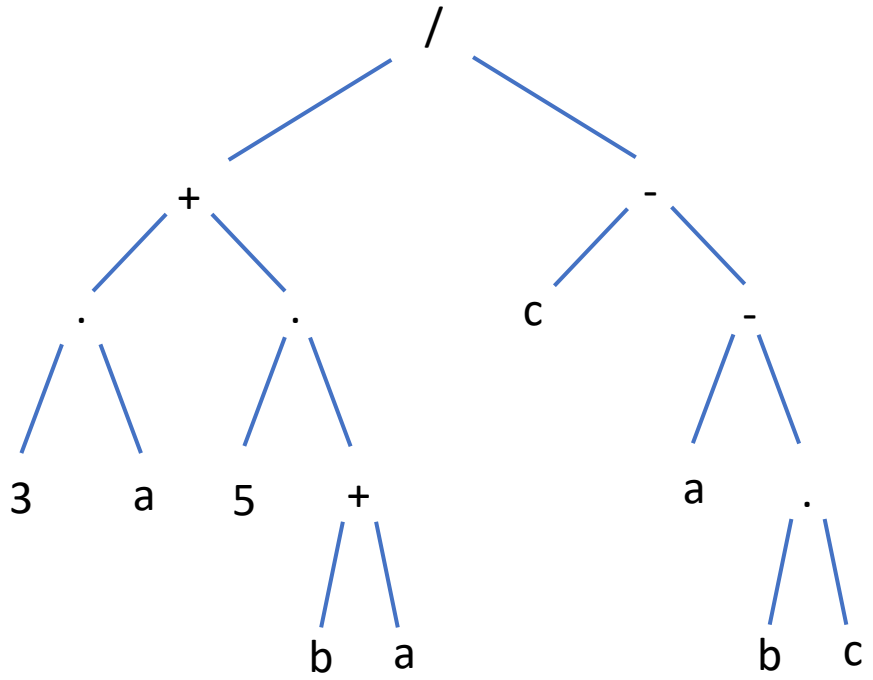
```
length([ H | T ], N) :- N = length(T) + 1.
```

Error: the predciate length(L,X) can be only true or false, but it does not have a numerical value, since predciates are **not** functions.

Arithmetic abstract syntax trees

Next week we start talking about grammars and parsing in application to English. However, these notions are fundamental for all programming languages. Take a course on the theory of compilation to learn more.

Here, we consider only a simple example: a syntax tree for arithmetical expressions. The binary syntax tree for $(3 \cdot a + 5 \cdot (b + a))/(c - (a - (b \cdot c)))$ is the following (ignore parentheses for simplicity):



Lesson to learn: arithmetic expressions can be represented uniquely as binary trees.

Arithmetic with addition and multiplication

For simplicity, let us focus on arithmetic expressions with addition and multiplication only. Use the term *sum*(*X*, *Y*) to represent the sum of *X* and *Y*. Use the term *prod*(*X*, *Y*) to represent the product of *X* and *Y*.

Last simplification: use one variable only. But any constants are fine.

Example: *sum*(*prod*(2, *X*), *prod*(4, 5)) means expression $((2 * X) + (4 * 5))$

Example: *prod*(*sum*(*X*, 7), *prod*(0, *sum*(*X*, 1))) means $((X + 7) * (0 * (X + 1)))$

Task: write a program that evaluates our simplified arithmetical expressions for a given number *N*. The predicate *evalPS*(*E*, *N*, *V*) is true if *V* is a value of an arithmetical expression *E*, when its single variable takes value *N*. *Do case analysis.*

```
evalPS(X,N,Val):- var(X), number(N), Val is N.  /*base case: variable*/
evalPS(X,N,Val):- number(X), Val is X.          /*base case: constant*/
```

Both *var*(*X*) and *number*(*X*) are library predicates. How to write recursive rules?

```
evalPS(sum(L,R),N, Value) :- evalPS(L,N,V1), evalPS(R,N,V2),
       Value is V1+V2.

evalPS(prod(L,R),N, Value) :- evalPS(L,N,V1), evalPS(R,N,V2),
       Value is V1 * V2.
```

Warning: only the first answer is correct, need "!" to cut backtracking. Example:
?- evalPS(sum(prod(A,5), prod(A,4)), 2,Val). returns Val=18

Recursion with arithmetic

Example: compute recursively the sum of a simple series

$$\sum_{k=1}^{N} k = 1 + 2 + 3 + 4 + \ldots + N.$$

Let predicate *seriesSum*(N, S) be true if S is the sum of the first N terms from this given series. To write a recursive program, do case analysis.

```
seriesSum(1,1).                                          /* base case */

seriesSum(N,S) :- N > 1, M is N-1,
                  seriesSum(M,X),        /*compute the previous sum recursively*/
                  S is X + N.                            /* add the last term */
```

Let the predicate *sumSquares*(N, S) be true if S is the sum of the first N squares:

$$\sum_{k=1}^{N} k^2 = 1^2 + 2^2 + 3^2 + 4^2 + \ldots + N^2.$$

```
sumSquares(1,1).                                         /* base case */
sumSquares(N,S) :- N > 1, M is N-1,
                  sumSquares(M,X),       /*compute the previous sum recursively*/
                  S is X + N^2.                          /* add the last term */
```

Summary

- You must be able to write queries to the knowledge bases
- You must be able to match lists
- You must be able to write recursive program
  - Recursion involving arithmetic operations
  - Recursion involving lists
  - Recursion involving (complex) terms
- You must be able to write programs that solve CSPs
  - Make sure you understand the difference between generate and test and interleaving generate and test
  - Why does interleaving help?

Minimize guessing: rule 1

Rule 1: Avoid guessing any value that is fully determined by other values and later testing if the guess is correct.

for example, instead of

```
uniq3(A,B,C),          % guess at A, B, and C
B is (A+C) mod 10      % then test if B is ok
```

we should use something like

```
uniq2(A,C),            % guess at A and C only
B is (A+C) mod 10      % calculate B once
uniq3(A,B,C)           % ensure they are all unique
```

The first version has a search space of $10 \times 10 \times 10 = 1000$.

The second has a search space of $10 \times 10 = 100$.

Minimize guessing: rule 2

Rule 2: Avoid placing independent guesses between the generation and testing of other values.

for example, instead of

```
dig(A), dig(B),      % guess at A and B
dig(C), dig(D),      % guess at C and D
A > B                % then test if A > B
```

we should use

```
dig(A), dig(B),      % guess at A and B
A > B,               % test if A > B
dig(C), dig(D),      % guess at C and D
```

In the first version, if we guess badly for A and B, we only get to reconsider *after* we have gone through *all* the values for C and D.

In the second version, if we guess badly for A and B, we reconsider immediately, before we consider any values for C and D.