YOU WANT PROOF?
I'LL GIVE YOU PROOF!
Sidney Harris is well-known for his cartoons about science, mathematics, and technology. He has drawn over 34,000 cartoons during his near 60-year career for magazines like *American Scientist*, *The New Yorker*, *Discover*, and *Science*. He lives in Brooklyn, New York.
Logic Programming

Logic is not the route to sleep.
Prolog Overview

• Introduction: goals, definitions, history

• Prolog programming: facts, queries, variables, rules, recursion

• Backward chaining (resolution)

• Data structures: functors, lists

• Unification

• Relation to logic: not a theorem prover, closed-world assumption

• * Relation to logic: horn clauses, skolemization, linear resolution
Class Goals

- Simple programming examples, lists
- *Not* practical programming: arithmetic, IO, real Prolog implementation, controlling search
- Underlying foundations: unification and resolution algorithms

Different goals for logic programming as opposed to functional programming.
Declarative Programming

Definition

A *declarative* programming languages means a language in which the programmer only specifies *what* to do and not *how* to do it.

To a (small) degree the SETL language exhibits declarative programming (cf. example of topological sort). Coming up we will use it as an example. SETL developed by J. Schwartz (NYU) around 1974.
A *non-procedural* programming language also means a language in which the programmer only specifies what to do and not how to do it.

But non-procedural sometimes taken to mean non-imperative in which case functional programming languages would have to be consider non-procedural.

So it is clearer to avoid the term *non-procedural* and use declarative or non-imperative depending on what is meant.
$ topological sort in SETL from
$ Schwartz, et al., Programming with Sets, page 408

$ the input G is a graph represented by a set of
$ ordered pairs; the output t is a total order
$ represented by an ordered tuple

proc top (G);
    nodes := (domain G) + (range G); $ set of all nodes
    t := []; $ initially nothing in order

    $ pick an n in nodes st n is not in range of G
    (while exists n in nodes | n notin range G)
    t with := n; $ n is next in total order
    G lessf := n; $ remove pairs with 1st elem n
    nodes less := n; $ remove n from nodes
    end while;
    return t;
end proc top;
The same program can be written in the (not declarative, but imperative) language Python using (not imperative, but functional) mechanism of list comprehension.
def rng(g):
    return {y for (_,y) in g}

def domain(g):
    return {x for (x,_) in g}

def top(g):
    nodes = domain(g) | range(g)
    t = []
    while True:
        ns = [n for n in nodes if n not in rng(g)]
        if ns == []:
            break
        t.append(ns[0])
        g = [(x,y) for (x,y) in g if x!=ns[0]]
        nodes.remove(ns[0])
    return t
SETL (like PROLOG) has a weak hold on being declarative.

\[\text{(while exists n in nodes | n notin range G)}\]

\[\text{ns = [n for n in nodes if n not in range(g)]}\]

The philosophical view in SETL is that the choice of \( n \) is independent of how it is implemented. Python on the other hand makes no pretense that finding the \( n \) is anything other than just an exhaustive, systematic search of the list.
In Star Trek: The Next Generation Captain Picard is often seen ordering tea from the replicator: “Tea, Earl Grey, Hot".
How, Not What

Some programmers would make a program like this to make tea.

Classical set theory and classical logic do not appear to hold any hope of eliminating the need of the programmer to be clever about how things are done.

Constructive logic, on the other hand, unifies the language of how and of what.

But that’s another story . . .
In the 1970s, with the work of Alan Colmerauer and Philippe Roussel of the University of Aix-Marseille in France and Robert Kowalski and associates at the University of Edinburgh in Scotland, researchers ... began to employ the process of logical deduction as a general-purpose model of computing.

Scott 4th, page 591.
Prolog is the quintessential member of the logic programming paradigm. To a degree Prolog (PROgrammation en LOGique) exhibits declarative programming.

- Logic programming – using logic to express what (not how) to compute and use searching for proofs to compute answers
Alain Colmerauer: 1941-2017

He was the center of a research group that, in the early 1970s, first conceived of Prolog (an abbreviation for “programmation en logique,” which is French for “programming in logic”), a general-purpose logic programming language rooted in first-order logic that is well-suited for tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates.

https://cacm.acm.org/news/
217533-in-memoriam-alain-colmerauer-1941-2017/fulltext
Information about Prolog


Information about Prolog

Stansifer in *The Study of Programming Languages*. 
Logic

For the second time (the first was axiomatic semantics) we need knowledge of basic first-order logic to proceed.
Logic

What is predicate logic?

Logic References

This topic is so important that references to additional reading is staggering.

- Classical Undergraduate Textbook: Suppes, ...
- On-line text: http://forallx.openlogicproject.org/
- Historical: Kneale
- Exhaustive: Handbook of Mathematical Logic (Barwise 1989)
- Ancient History:
  - Computer Science: Reeves
  - CS and Theorem Proving: Gallier
  - CS and Model Theory: Huth and Ryan
Recap of First-Order Logic

\[
\begin{align*}
\bot & \quad \text{false} \\
\top & \quad \text{true} \\
A \& B & \quad A \text{ and } B \\
A \lor B & \quad A \text{ or } B \\
\neg A & \quad \text{not } A \\
A \Rightarrow B & \quad A \text{ implies } B \\
\forall x \ P(x) & \quad \text{for all } x, \ P(x) \\
\exists x \ P(x) & \quad \text{there exists } x, \ P(x)
\end{align*}
\]

*Modus ponens*, one of the classic laws of deduction

\[
\begin{array}{c}
A \Rightarrow B \\
\hline
A \\
\hline
B
\end{array}
\]
1.2 Natural deduction

The basic rules of natural deduction:

<table>
<thead>
<tr>
<th>Introduction</th>
<th>Elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \land )</td>
<td>( \frac{\phi \land \psi}{\phi \land \psi} ) ( \land_i )</td>
</tr>
<tr>
<td></td>
<td>( \frac{\phi \land \psi}{\psi} ) ( \land e_2 )</td>
</tr>
<tr>
<td>( \phi \land \psi )</td>
<td>( \phi \lor \psi )</td>
</tr>
</tbody>
</table>
Logic

Logic formalizes sentences and reasoning (from Grimaldi, Example 2.13, page 68). Let $p$, $q$ denote the primitive statements:

$p$: Joan goes to Lake George.
$q$: Mary pays for Joan’s shopping spree.

and consider the implication

$p \implies q$: If Joan goes to Lake George, then Mary will pay for Joan’s shopping spree.

How does one write in clear English the negation of $p \implies q$?

$p \land \neg q$: Joan goes to Lake George, but Mary does not pay for Joan’s shopping spree.
Logic

Logic is not about empirical, real-world truth. It is about argument, structure, and reasoning, an invisible process with good properties (e.g., relevance to the “real” world).

Consider these examples (from Reeves and Clarke):

Paris is in Australia and Australia is below the equator. 
Therefore, Paris is in Australia.

Valid argument, wrong geography.

The Eiffel Tower is in Paris or Paris is in France. 
Therefore, the Eiffel Tower is in Paris.

Invalid argument, correct geography.
Logic: Valid Argument (Lewis Carroll)

- $\forall x(D(x) \Rightarrow \neg W(x))$ “All ducks are unwilling to waltz.”
- $\forall x(F(x) \Rightarrow W(x))$ “All officers are willing to waltz.”
- $\forall x(P(x) \Rightarrow D(x))$ “All my poultry are ducks.”

Therefore $\forall x(P(x) \Rightarrow \neg F(x))$ “All my poultry are not officers.”

Since the statements are all universally quantified, a propositional proof suffices. $Pa \Rightarrow Da \Rightarrow \neg Wa \Rightarrow \neg Fa$. The argument is correct and relies on the (classical) contrapositive of the second premise.
All the dated letters in this room are written on blue paper.  \( ID \Rightarrow BP \)

None of them are in black ink, except those that are written in the third person.  \( BI \Rightarrow TP \)

I have not filed any of those that I can read.  \( CR \Rightarrow \neg F \)

None of those that are written on one sheet are undated.  \( WO \Rightarrow ID \)

All of those that are not crossed out are in black ink.  \( \neg CO \Rightarrow BI \)

All of those that are written by Brown begin with “Dear Sir.”  \( BB \Rightarrow DS \)

All of those that are written on blue paper are filed.  \( BP \Rightarrow F \)

None of those that are written on more than one sheet are crossed out.  \( \neg WO \Rightarrow \neg CO \)

None of those that begin with “Dear Sir” are written in the third person.  \( DS \Rightarrow \neg TP \)

Therefore, I cannot read any of Brown’s letters \( BB \Rightarrow \neg CR \). Is the argument valid?
Formalization

Propositional logic suffices as all the statements are universally quantified. Ten true/false propositions are required which we name along with a hint as to their intended meaning as follows:

- **ID** Is dated
- **BP** On blue paper
- **BI** In black ink
- **TP** In the third person
- **F** Is filed
- **CR** Can read
- **WO** Written on one sheet
- **CO** Crossed out
- **BB** By Brown
- **DS** Begins with “Dear Sir”
Validity Proof

Suppose a letter is written by Brown $BB$. By *modus ponens* and *tollens*:

6 All of those that are written by Brown begin with “Dear Sir.” $BB \Rightarrow DS$, so $DS$ [mp].

9 None of those that begin with “Dear Sir” are written in the third person. $DS \Rightarrow \neg TP$, so $\neg TP$ [mp].

2 None of them are in black ink, except those that are written in the third person. $BI \Rightarrow TP$, so $\neg BI$ [mt].

5 All of those that are not crossed out are in black ink. $\neg CO \Rightarrow BI$, so $CO$ [mt].

8 None of those that are written on more than one sheet are crossed out. $\neg WO \Rightarrow \neg CO$, so $WO$ [mt].

4 None of those that are written on one sheet are undated. $WO \Rightarrow ID$, so $ID$ [mp].

1 All the dated letters in this room are written on blue paper. $ID \Rightarrow BP$, so $BP$ [mp].

7 All of those that are written on blue paper are filed. $BP \Rightarrow F$, so $F$ [mp].

3 I have not filed any of those that I can read. $CR \Rightarrow \neg F$, so $\neg CR$ [mt].

Therefore, I cannot read any of Brown’s letter $BB \Rightarrow \neg CR$ (by implication introduction).
The Logic Daemon responds...

Sequent attempted:  
\[ B \rightarrow D, D \rightarrow T, I \rightarrow T, \neg C \rightarrow I, \neg W \rightarrow C, W \rightarrow D, D \rightarrow P, P \rightarrow F, R \rightarrow F | B \rightarrow R \]

<table>
<thead>
<tr>
<th>OK</th>
<th>1</th>
<th>(1)</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>2</td>
<td>(2)</td>
<td>B \rightarrow D</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2</td>
<td>(3)</td>
<td>D</td>
<td>1,2 \rightarrow E</td>
</tr>
<tr>
<td>OK</td>
<td>4</td>
<td>(4)</td>
<td>D \rightarrow T</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4</td>
<td>(5)</td>
<td>\neg T</td>
<td>3,4 \rightarrow E</td>
</tr>
<tr>
<td>OK</td>
<td>6</td>
<td>(6)</td>
<td>I \rightarrow T</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4,6</td>
<td>(7)</td>
<td>\neg I</td>
<td>6,5\text{MTT}</td>
</tr>
<tr>
<td>OK</td>
<td>8</td>
<td>(8)</td>
<td>\neg C \rightarrow I</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4,6,8</td>
<td>(9)</td>
<td>C</td>
<td>8,7\text{MTT}</td>
</tr>
<tr>
<td>OK</td>
<td>10</td>
<td>(10)</td>
<td>\neg W \rightarrow C</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4,6,8,10</td>
<td>(11)</td>
<td>W</td>
<td>10,9\text{MTT}</td>
</tr>
<tr>
<td>OK</td>
<td>12</td>
<td>(12)</td>
<td>W \rightarrow D</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4,6,8,10,12</td>
<td>(13)</td>
<td>D</td>
<td>12,11 \rightarrow E</td>
</tr>
<tr>
<td>OK</td>
<td>14</td>
<td>(14)</td>
<td>D \rightarrow P</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4,6,8,10,12,14</td>
<td>(15)</td>
<td>P</td>
<td>14,13 \rightarrow E</td>
</tr>
<tr>
<td>OK</td>
<td>16</td>
<td>(16)</td>
<td>P \rightarrow F</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4,6,8,10,12,14,16</td>
<td>(17)</td>
<td>F</td>
<td>16,15 \rightarrow E</td>
</tr>
<tr>
<td>OK</td>
<td>18</td>
<td>(18)</td>
<td>R \rightarrow F</td>
<td>A</td>
</tr>
<tr>
<td>OK</td>
<td>1,2,4,6,8,10,12,14,16,18</td>
<td>(19)</td>
<td>\neg R</td>
<td>18,17\text{MTT}</td>
</tr>
<tr>
<td>OK</td>
<td>2,4,6,8,10,12,14,16,18</td>
<td>(20)</td>
<td>B \rightarrow R</td>
<td>19 \rightarrow I(1)</td>
</tr>
</tbody>
</table>

Congratulations. Your proof is correct.

http://logic.tamu.edu/daemon.html
In formalizing the argument in PROLOG, we have a problem. There is no “not” in PROLOG, and so PROLOG cannot use the contrapositive of a rule even though it logically equivalent.

We cobble together a solution by introducing predicate symbols for the negation of each predicate and by asserting both the implication and its contrapositive.

OnBlue :- IsDated. (* [1a] IsDated => OnBlue *)
NotDated :- NotOnBlue. (* [1b] NotOnBlue => NotDated *)

InThird :- InBlack. (* [2a] InBlack => InThird *)
NotInBlack :- NotInThird. (* [2b] NotInThird => NotInBlack *)

NotFiled :- CanRead. (* [3a] CanRead => NotFiled *)
NotCanRead :- IsFiled. (* [3b] IsFiled => NotCanRead *)
IsDated :- OneSheet.  (* [4a] OneSheet => IsDated *)
NotOneSheet :- NotDated. (* [4b] NotDated => NotOneSheet *)

InBlack :- NotCrossedOut. (* [5a] NotCrossedOut => InBlack *)
CrossedOut :- NotInBlack. (* [5b] NotInBlack => CrossedOut *)

DearSir :- ByBrown.  (* [6a] ByBrown => DearSir *)
NotByBrown :- NotDearSir. (* [6b] NotDearSir => NotByBrown *)

IsFiled :- OnBlue.  (* [7a] OnBlue => IsFiled *)
NotOnBlue :- NotFiled. (* [7b] NotFiled => NotOnBlue *)

NotCrossedOut:-NotOneSheet.(* [8a] NotOneSheet => NotCrossedOut *)
OneSheet :- CrossedOut.  (* [8b] CrossedOut => OneSheet *)

NotInThird :- DearSir.  (* [9a] DearSir => NotInThird *)
NotDearSir :- InThird. (* [9b] InThird => NotDearSir *)
Logic: Valid Argument (PROLOG) 1

Only by picking the correct form of the premises can we verify the argument in PROLOG.

(* ByBrown => NotCanRead? *)

ByBrown.
NotCanRead?

NotCanRead :- IsFiled 3b
    :- OnBlue 7a
    :- IsDated 1a
    :- OneSheet 4a
    :- CrossedOut 8b
    :- NotInBlack 5b
    :- NotInThird 2b
    :- DearSir 9a
    :- ByBrown 6a
Logic: Valid Argument (PROLOG) 2

Only by picking the correct form of the premises can we verify the argument in PROLOG.

(* CanRead => NotByBrown? *)
CanRead.
NotByBrown?

NotByBrown :- NotDearSir 6b
  :- InThird 9b
  :- InBlack 2a
  :- NotCrossedOut 5a
  :- NotOneSheet 8a
  :- NotDated 4b
  :- NotOnBlue 1b
  :- NotFiled 7a
  :- CanRead 3a
Logic

Exercise 12 (from Grimaldi, page 98). Write each of the following arguments in symbolic form. Then establish the validity of the argument or give a counterexample to show that it is invalid.

*If Rachael gets the supervisor’s position and works hard, then she’ll get a raise. If she gets the raise, then she’ll buy a new car. She has not purchased a new car. Therefore either Rachel did not get the supervisor’s position or she did not work hard.*

\[ s \land w \Rightarrow r \]
\[ r \Rightarrow c \]
\[ \neg c \]
\[ \therefore \neg s \lor \neg w \]

Valid: *modus tollendo tollens* twice, and DeMorgan’s Law
If Dominic goes to the racetrack, then Helen will be mad. If Ralph plays cards all night, then Carmela will be mad. If either Helen or Carmela gets mad, then Veronica (their attorney) will be notified. Veronica has not heard from either of these two clients. Consequently, Dominic didn't make it to the racetrack and Ralph didn't place cards all night.

\[ r \Rightarrow h \]
\[ d \Rightarrow c \]
\[ h \lor c \Rightarrow v \]
\[ \neg v \]
\[ \therefore \neg r \& \neg d \]
Describe in words the rules of inference:

1. Modus Ponens
2. Modus Tollendo Tollens
3. DeMorgan’s laws (equivalences)
4. Conjunction introduction
5. Conjunction elimination

Use them to prove

\[ r \Rightarrow h \]
\[ d \Rightarrow c \]
\[ h \mid c \Rightarrow v \]
\[ \neg v \]
\[ \therefore \neg r \land \neg d \]
R→H, D→C, (HvC)→V, ~V |- ~R&~D

1 (1) ~V A
2 (2) (HvC)→V A
1,2 (3) ~(HvC) 1,2MTT
1,2 (4) ~H&~C 3DM
1,2 (5) ~H 4&E
1,2 (6) ~C 4&E
7 (7) R→H A
1,2,7 (8) ~R 5,7MTT
9 (9) D→C A
1,2,9 (10) ~D 6,9MTT
1,2,7,9 (11) ~R&~D 8,10&I
Ontology:
• objects, called atoms, and
• relationships between objects.

Interactive dialog:
• declare facts about objects
• define rules about relationships and objects
• ask questions
Prolog Programming

Valuable (Gold). /* Gold is valuable. */
Valuable (Money). /* Money is valuable. */
Father (John, Mary). /* John is the father of Mary. */
Gives (John, Book, Mark). /* John gives the book to Mark. */
King (John, France). /* John is the king of France. */
Iam. /* I am. */

Comments: /* ... */.

Chair. /* Nonsense (but syntactically legal). */
Bananna (Speaks). /* More nonsense. */

Semantically, nouns do not make good predicate names. *Predicate*: the part of a sentence that expresses what is said of the subject and usually consists of a verb with or without objects.
**Syntax Convention**

**Warning:** non-standard syntax in use here. (Why? Because mine is simpler, more natural, several different conventions in use anyway, and emphasizes the point that some convention about identifiers is crucially important.)

- Upper case: predicate symbols, functors, and atoms are in upper case.
- Lower case: variables are in lower case.
- Period: Facts and rules end in a period.
- Question mark: Queries end in a question mark.
Prolog Programming

Father (John, Mary). /* These two facts are different, */
Father (Mary, John). /* order of arguments matters. */

The list of asserted facts comprises a database over which we may pose questions. Queries are literals, just as facts are. They are true when a matching fact is found in the database; Prolog doesn’t know anything.
Queries

Assuming the 6 original facts ...

Valuable (Gold)? /* Is Gold valuable? */ yes
Valuable (Money)? /* Is Money valuable? */ yes
Valuable (Diamonds)? /* Are Diamonds valuable?! */ no
Father (John, Mary)? yes
Father (Mary, John)? no
Father (John, Anne)? no
Greek (Socrates)? /* Prolog is not an oracle. */ no

If a matching fact is found in the database, Prolog responds with yes; otherwise no.
Variables in Queries

Atoms are not the only terms. Variables can appear in literals too. Prolog treats variables in queries as unknowns and tries to find objects, which, when substituted for the variables, give a literal that appears in the list of facts.

Father (x, Mary)? /* Who is the father of Mary? */
  x=John
Father (John, x)? /* Whom is John the father of? */
  x=Mary
Father (x, Karen)? /* Who is the father of Karen? */
  no

The response x= something, does not indicate an association lasting into the future as does assignment in imperative languages. There can be more than one choice for the variables that results in a literal in the database.

Valuable (x)? /* What is valuable? */
  x=Gold
  x=Money
Variables in Queries

If there is more than one variable in the query, a Prolog response contains a binding for each.

Gives (who, what, Mark)?
who=John, what=Book

It is not possible to use variables in the relationship position.

x (John, Mary)? /* What’s their relationship? */
Real Interaction in Prolog

% pl
Welcome to SWI-Prolog (Version 2.9.10)
Copyright (c) 1993-1997 University of Amsterdam. All rights reserved.

1 ?- [puppy].
puppy compiled, 0.01 sec, 5,784 bytes.

Yes
2 ?- puzzle(X,Y,Z,W).

X = puppy(lauren, trouper, collie)
Y = puppy(nicky, happy, terrier)
Z = puppy(robin, wiggles, reitreiver)
W = puppy(terry, frisky, poodle) ;

No
Conjunction

To find out if two or more facts are true simultaneously, queries are permitted to have a list of literals separated by commas.

Gives (John, x, Mark), Valuable (x)?

A list of literals acts like a conjunction. The variable $x$ means the same throughout the query, but has no connection with any $x$ appearing in any other query. The response to this query is negative because there is no object $x$ such that both the literals Gives(John,x,Mark) and Valuable(x) are in the database. This is different than the two separate queries:

Gives (John, x, Mark)?
- $x$=Book

Valuable (x)?
- $x$=Gold
- $x$=Money
This is different still than using two different variables:

Gives (John, x, Mark), Valuable (y)?
- x=Book, y=Gold
- x=Book, y=Money
Variables can appear in assertions as well as queries, though this is rarer. Variables stand for all objects.

Beautiful (x). /* Everything is beautiful. */

Beautiful (Butterfly)? /* Is a butterfly beautiful? */ yes

Beautiful (z)? /* Is everything beautiful? */ yes

Why isn’t there a binding to z in the solution to the last query? Any binding to the variable z works. So, we omit this binding and act as if the variable z did not appear in the query. For the sake of simplicity, let us agree to call this just one solution, though z=Butterfly, z=Me, z=You, and infinitely many more bindings are legitimate answers.
What are variables?
Variables stand for any value.

Variables can be renamed without changing the meaning. The variables in a query are not related to variables in any rule (even if they happen to have the same name).

In other words, the scope of a variable is just the single clause it appears in.
Responses

Each solution found by Prolog gets a response. Typically, for single response each variable in the query gets a binding. (These bindings will be called a “substitution” later.) If there are no variables in the query, then the response is just “yes.” If a binding to a variable is to a variable (not a term), then the binding can be omitted. Actually Prolog usually prints a binding to an internal variable name. Something like:

\[ z = _x435 \]
Responses

No solutions

no

One solution; no variables in the query (or all bindings are trivial)

yes

Two solutions; no variables in the query (or all bindings are trivial)

yes

yes

Three solutions; at least the variables $x$ and $y$ appear in the query.

$x = A, \ y = M$

$x = B, \ y = M$

$x = C, \ y = M$
Conditionals

Conjunction is not needed for assertions, but conditional assertion is. Assertions of this kind are called *rules*. Rules have two parts: the head and tail separated by a symbol “:-” called a *turnstile*.

Iam :- Ithink. /* If I think, then I am. */
§ 2. A judgment will always be expressed by means of the sign

\[ \vdash \]

which stands to the left of the sign, or the combination of signs, indicating the content of the judgment.
Quantifiers

Using variables and implication in Prolog we can capture the classic syllogism:

\[
\text{Mortal (x)} :\text{- Man (x).} \quad /* \text{All men are mortal.} \quad */ \\
\text{Man (Socrates).} \quad /* \text{Socrates is a man.} \quad */ \\
\text{Mortal (Socrates)}? \\
yes
\]
Example from Scott 6th page 595.

Takes (Jane_Doe, HUM2011).
Takes (Jane_Doe, CSE2054).
Takes (Ajit_Chandra, LNG3110).
Takes (Ajit_Chandra, CSE2054).

Classmates (x,y) :- Takes(x,z), Takes(y,z).

Classmates (Jane_Doe, x)?
x = Ajit_Chandra
Quantification

Takes (Jane_Doe, HUM2011).
Takes (Jane_Doe, CSE2054).
Takes (Jane_Doe, LNG3110).
Takes (Ajit_Chandra, LNG3110).
Takes (Ajit_Chandra, CSE2054).

Classmates (x,y) :- Takes(x,z), Takes(y,z).

Classmates (Jane_Doe, x)?
x = Ajit_Chandra
x = Ajit_Chandra

Two distinct proofs.
Takes (Jane_Doe, HUM2011).
Takes (Jane_Doe, CSE2054).
Takes (Ajit_Chandra, LNG3110).
Takes (Ajit_Chandra, CSE2054).
Takes (Juan_Pérez, HUM2011).
Takes (Juan_Pérez, MTH4001).

Classmates (x,y) :- Takes(x,z), Takes(y,z).

Classmates (Jane_Doe, x)?
  x = Ajit_Chandra
  x = Juan_Pérez
A rule that defines the paternal grandfather relation:

Grandfather(c,g) :- Parents(c,m1,f), Parents(f,m2,g).

Note the different uses of the variables in this rule. The variables g and c appear in the head of the clause have the effect of making this rule a definition of the grandfather relation in general, rather than for specific cases. The variable f is used to form a connection between the two Parents literals. The variables m1 and m2 are “don’t care” variables. Some Prolog systems permit variables used like this to be replaced with a special anonymous variable symbol such as _.
From

Parents (George, Alexandra, Edward).
Parents (Edward, Victoria, Albert).

Prolog can conclude that Grandfather (George, Albert).
Choice, Recursion

Grandfather (c,g) :- Parents (c,m1,f), Parents (f,m2,g).
Grandfather (c,g) :- Parents (c,m1,f), Parents (m1,m2,g).

Prolog tries all possible ways to establish a literal, so both definitions are employed. Hence two rules with the same head act like choices.
It is possible and useful to define relations in Prolog in terms of themselves.

Ancestor (c,a) :- Parents (c,m,a).
Ancestor (c,a) :- Parents (c,a,f).
Ancestor (c,a) :- Parents (c,m,f), Ancestor (m,a).
Ancestor (c,a) :- Parents (c,m,f), Ancestor (f,a).
Example

1 In(Atlanta,Georgia).
2 In(Houston,Texas).
3 In(Austin,Texas).
4 In(Toronto,Ontario).
5 In(x,USA) :- In (x,Georgia).
6 In(x,USA) :- In (x,Texas).
7 In(x,Canada) :- In (x,Ontario).
8 In(x,NA) :- In (x,USA).
9 In(x,NA) :- In (x,Canada).
Example

In (Atlanta, NA)? /* Is Atlanta in NA? */

The search process takes the following steps:

- **Goal:** In (Atlanta, NA)
  Matches rule (8): In(x,NA) if x=Atlanta
- **Goal:** In (Atlanta, USA)
  Matches rule (5): In(x,USA) if x=Atlanta
- **Goal:** In (Atlanta, Georgia)
  Matches fact (1). Success! No more goals.
Example

In (Austin, NA)?

The search process takes the following steps:

- **Goal:** In (Austin, NA)
  Matches rule (8): \( \text{In}(x, \text{NA}) \) if \( x = \text{Austin} \)

- **Goal:** In (Austin, USA)
  Matches rule (5): \( \text{In}(x, \text{USA}) \) if \( x = \text{Austin} \)

- **Goal:** In (Austin, Georgia)
  No rule matches! Backtrack.

- **Goal:** In (Austin, USA)
  Matches rule (6): \( \text{In}(x, \text{USA}) \) if \( x = \text{Austin} \)

- **Goal:** In (Austin, Texas)
  Matches fact (3). Success! No more goals.
In (Toronto, NA)? /* Is Toronto in NA? */

• Goal: In (Toronto, NA)
  Matches rule (8): In(x,NA) if x=Toronto

• Goal: In (Toronto, USA)
  Matches rule (5): In(x,USA) if x=Toronto

• Goal: In (Toronto, Georgia)
  No rule matches! Backtrack.

• Goal: In (Toronto, USA)
  Matches rule (6): In(x,USA) if x=Toronto

• Goal: In (Toronto, Texas)
  No rule matches! Backtrack.

• Goal: In (Toronto, NA)
  Matches rule (9): In(x,NA) if x=Toronto

• Goal: In (Toronto, Canada)
  Matches rule (7): In(x,Canada) if x=Toronto

;
Example from Clocksin and Mellish.

1 Male (Albert).
2 Male (Ed).
3 Female (Alice).
4 Female (Victoria).
5 Parents (Ed, Victoria, Albert).
6 Parents (Alice, Victoria, Albert).
7 Sister(x,y) :- Female(x),Parents(x,m,f),Parents(y,m,f).
Sister (Alice, Ed)? /* Is Ed the sister of Alice? */

The search process takes the following steps:

- **Goal: Sister (Alice, Ed)**
  Matches rule (7): Sister (x,y) if x=Alice, y=Ed

- **Goal: Female(Alice), Parents(Alice,m,f), Parents(Ed,m,f)**
  Matches fact (3): Female (Alice)

- **Goal: Parents(Alice,m,f), Parents(Ed,m,f)**
  Matches fact (6): Parents (Alice, Victoria, Albert) if m=Victoria, f=Albert

- **Goal: Parents (Ed, Victoria, Albert)**
  Matches fact (5). Success! No more goals.
Sister (Alice, x)? /* Who are the sisters of Alice? */

The search process takes the following steps:

- **Goal:** Sister (Alice, xₙ)
  Matches rule (7): Sister(x₂,y) if x₂=Alice, and x₁=y

- **Goal:** Female(Alice), Parents(Alice,m,f), Parents(y,m,f)
  Matches fact (3): Female (Alice)

- **Goal:** Parents(Alice,m,f), Parents(y,m,f)
  Matches fact (6): Parents (Alice, Victoria, Albert) if m=Victoria, f=Albert

- **Goal:** Parents(y,Victoria,Albert)
  Matches fact (5): Parents (Ed, Victoria, Albert) if y=Ed

No clauses remain, so the search succeeded. Tracing through the execution we conclude that x=x₁=y=Ed, so x=Ed.
Example

What if we reject the choice of fact 5 and force backtracking. Are there other solutions?
Actually, the very next fact is another match and we get another solution, namely $x = \text{Alice}$.

- **Goal:** Parents($y$, Victoria, Albert)
  Matches fact (6): Parents (Alice, Victoria, Albert) if $y = \text{Alice}$

So, by this definition Alice is her own sister.
We would like to revise the definition

Sister($x$, $y$) :- Female($x$), Parents($x$, $m$, $f$),
Parents($y$, $m$, $f$), $x \neq y$

But it is not at all clear what $x \neq y$ means.
Backward Chaining

The algorithm seeks to establish a list of goals in a database of facts or rules. This search strategy is sometimes called \textit{backward chaining} because it starts from the goals and works backward to justify them instead of starting from the premises and working forward to find the goals.

1. If list of goals is empty, then success.
2. Match the first goal in the list with the some head of some rule in the database. If none such, then backtrack. OW, replace goal with tail of rule.

Note:

- Search the database in order.
- New goals go at the head of the list.

Success could be found zero, one, or more ways.
Backward Chaining (All Solutions)

solve :: Eq a => [[a]] -> [[a]] -> [a] -> [[[a]]]
solve db [] = [(())]  
{- success! -}
solve db (g:gs) = concatMap (solve db) [ t++gs | (h:t) <- db],

(Ignoring unification.) Find all solutions in a dfs of search space. Lazy evaluation yields the first solution without evaluating the others!
Backward Chaining (All Solutions)

solve db [] = [[]]  \{- success! -\}
solve db (g:gs) = concatMap (solve db) [ t++gs | (h:t) <- db, g==h ]

solve :: Eq a => [[a]] -> [[a]] -> [a] -> [[[a]]]
solve db path [] = [path]  \{- success! -\}
solve db path (g:gs) = concatMap (solve db (path++[g:gs])) [ t++gs | (h:t) <- db, g==h ]

solveBFS db [] = []  \{- failure! -\}
solveBFS db (([]:gs):paths) = [reverse gs]  \{- success! -\}
solveBFS db ((g:gs):paths) =

(Ignoring unification.) Find all solutions in a dfs of search space. Lazy evaluation yields the first solution without evaluating the others!
Backward Chaining (All Solutions)

```
solve db [] = [[]]  \{- success! -\}
solve [] goals = []  \{- fail! -\}
solve db (g:gs) = tryG db
  where
    tryG [] = []  \{- No further matches \-
    tryG ((h:t): rest)
        | g==h = solve db (t++gs) ++ (tryG rest)
        | otherwise = tryG rest

(Ignoring unification.) Find all solutions in a dfs of search space. Lazy evaluation yields the first solution without evaluating the others!
```
Backward Chaining (All Solutions)

db = ['ABCD', 'AEF', 'BF', 'E', 'F', 'AF']

solve db "AE"
-- goals A,E
tryG db "AE"
n (solve db ("BCD" ++ "E") ++ (tryG ['AEF', 'BF', 'E', 'F', 'AF']
  (solve db "BCDE") ++ (tryG ['AEF', 'BF', 'E', 'F', 'AF']
-- goals B,C,D,E
  (tryG ['BF', 'E', 'F', 'AF'] "BCDE") ++ (tryG ['AEF', 'BF', 'E', 'F', 'AF']
db = ['ABCD', 'AEF', 'BF', 'E', 'F', 'AFA']
xkcd: Depth and Breadth

-- nx :: ID -> [ID] all the subsequent IDs
-- queue
solve nx [] = false
solve nx (c:cs) = (final c) || (solve nx (cs ++ (nx c)))

solve :: (b -> [b]) -> [b] -> Bool
solve nx cs = any final cs || solve nx (concatMap nx cs)

-- For monads like list and Data.Sequence.Seq
solve :: (Foldable m, Monad m) => (b -> m b) -> m b -> Bool
solve nx cs = any final cs || solve nx (cs >>= nx)
What is the difference between a tree and a traversal of a tree?

The *search space* for a query is a tree; the Prolog system performs a traversal of the search space.

The order in which you draw the entire search space is immaterial. The method of traversing the search space is crucial.

Prolog does not first create the entire search space and then traverse it. But one cannot understand Prolog theoretically without being able to visualize the entire search space of a Prolog program.
A nullary predicate is a predicate with zero arguments, e.g., Iam versus Gives(x,y,z).

Nullary predicates are rare in practice, but they allow us to focus on the creation of the Prolog search space and let us defer the discussion about unifying objects until later.
Search Space Examples

Consider the query A,E? in the following database of facts and rules about nullary predicates A, B, C, D, E, and F.

1  A :- B,C,D.
2  A :- E,F.
3  B :- F.
4  E.
5  F.
6  A :- F.

Original
The order in which one creates nodes in the search space does not matter if the entire search space is to be contemplated abstractly. Since the search space is large or possibly infinite, a systematic order in creating the nodes is desirable. There are two natural ways to search the search space: breadth-first search and depth-first search.
Create the search space
Breadth-first
Create the search space
Depth-first
Observe that the order of the answers differ.
Consider the query \( A, E? \) in a variation of the original database of facts and rules. Modify rule six slightly.

Original

1. \( A : - B, C, D. \)
2. \( A : - E, F. \)
3. \( B : - F. \)
4. \( E. \)
5. \( F. \)
6. \( A : - F. \)

Variation I

1. \( A : - B, C, D. \)
2. \( A : - E, F. \)
3. \( B : - F. \)
4. \( E. \)
5. \( F. \)
6. \( A : - F, A. \)
The Prolog search space is always a *tree*. But it can be an *infinite* tree. Sometimes it can be convenient to represent an infinite tree as a graph (the dfs tree with back edges).
Infinite Tree

A, E

1

B, C, D, E

E, F, E

F, A, E

1 2 3 4 5 6

C, D, E

E

F, E

F, C, D, E

© 14 October 2022
Modify the previous database of facts and rules by switching rules 1 and 6. Consider the same query A,E?.

<table>
<thead>
<tr>
<th>Variation I</th>
<th>Variation II</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 A :- B,C,D.</td>
<td>1 A :- F,A.</td>
</tr>
<tr>
<td>2 A :- E,F.</td>
<td>2 A :- E,F.</td>
</tr>
<tr>
<td>3 B :- F.</td>
<td>3 B :- F.</td>
</tr>
<tr>
<td>4 E.</td>
<td>4 E.</td>
</tr>
<tr>
<td>5 F.</td>
<td>5 F.</td>
</tr>
<tr>
<td>6 A :- F,A.</td>
<td>6 A :- B,C,D.</td>
</tr>
</tbody>
</table>
Changing the order of the rules will only reorder the branches of the search space. The search space remains substantially the same. In particular, the size and the “logic” is the same.
Search Space Examples

Consider the query \( C(a), B(x) \) in the following database of facts and rules about binary predicates \( A, B, \) and \( C \).

1. \( A(y) :- C(y). \)
2. \( B(x) :- A(d). \)
3. \( B(x) :- C(c). \)
4. \( C(a) :- B(b). \)
5. \( C(d). \)
Search Space Examples

\[
C(a), B(x)
\]
\[
B(b), B(x)
\]
\[
*A(d), B(x) + C(c), B(x)
\]
\[
C(d), B(x) B(c), B(x)
\]
\[
B(x) + C(c), B(x) * A(d), B(x)
\]
\[
@A(d) !C(c)
\]
\[
C(d) B(c)
\]
\[
[ ] @A(d) !C(c)
\]
asserta/1, assertz/1, call/1, !/0 (cut), get_char/1, unify_with_occurs_check/2, var/1, etc.
Fail and Cut

fail and cut (!) are used to control the search.

Bird (Eagle).
Bird (Sparrow).
Bird (Penguin).
Fly (Penguin) :- !, fail. /* Except penguins...*/
Fly (x) :- Bird (x). /* ... all birds fly.*/

Consider the query B? in the following databases of facts and rules.

1  A :- C,! , C.
2  A.
3  B :- A.
4  B :- C.
5  C.
6  C :- D.
Cut is particularly useful in a programming strategy in Prolog called *generate and test*. In programs that use the generate-and-test strategy the goal consists of subgoals that generate potential solutions, which are then check by later “test” subgoals. Rejected solutions require backtracking to “generate” subgoals.

Sebesta 15th, page 715.

The purpose of cut is to stop the search when the one solution is found.
Cut is necessary to control search and makes a lie of the claim of declarative programming.
Brute force search is only occasionally a useful strategy. Take the generate and test way to sort.

Sort (xs, ys) :- Permutation (xs, ys), Sorted(ys).

Permutation ([], []).
Permutation (l, [h|t]) :-
    Append (v, [h|u], l), Append(v, u, w), Permutation (w, t).

Sorted([]).
Sorted([x]).
Sorted([x, y|rest]) :- Precedes(x, y), Sorted ([y|rest])
She’s a witch! (at YouTube)

A scene from *Monty Python and the Holy Grail*

Villager: We have found a witch, may we burn her?
Crowd: BURN!!! BUUUURN HER!
Bedever: But how do you know she is a witch?
Villager: She look like one!
Consider the following assumptions used by Sir Bedever in *Monty Python and the Holy Grail* to argue that the girl is a witch.

Witch (x) :- Burns (x).
Burns (x) :- MadeOfWood (x).
MadeOfWood (x) :- Floats (x).
Floats (x) :- SameWeight (x, Duck).
SameWeight (Girl, Duck).

Witch (Girl)?
The argument is valid.

Witch (Girl)?
Burns (Girl)?
MadeOfWood (Girl)?
Floats (Girl)?
SameWeight(Girl,Duck)?

✓

Girl: That’s a fair cop.
Witch (x) :- Burns (x).
Burns (x) :- MadeOfWood (x).
MadeOfWood (x) :- Floats (x).
Floats (x) :- SameWeight (x, y), Floats (y).
SameWeight (Girl, Duck).

Witch (Girl)?
We divided understanding Prolog into two important parts:

- Finding a solution (resolution)
- Matching (unification)

Resolution starts with the query as the goal, then considers each of the Prolog facts and rules in order, and replaces the first goal with the list of its prerequisites. This leads to a search space that arises from the Prolog program. We have glossed over a subroutine of this process, namely the matching of the first goal with the head of a rule. It is time to investigate this algorithm more carefully. However, First we generalize objects in Prolog to include structured objects called functors. Then later we introduce the unification problem and unification algorithms.
Thus far it does not seem possible that Prolog could be used for ordinary programming.

- Where are functions?
- Where is data?
- What about modularity?

We take up the first two questions and completely ignore the third.
Functions

Where are functions in Prolog?
How does one compute anything?

Functions are a kind of relation!

Not very intuitive, but nothing needs to be added to the language.
Data Structures

Where are data structures in Prolog?

Functors construct data

We now add them to the language.
Data structures are achieved in Prolog using functors.

**Definition**

A *functor* is an uninterpreted function symbol that construct objects from arguments.

Here are two complex objects/values constructed using functors Book and Author:

Book (WutheringHeights, Author (Emily, Bronte))
Book (Ulysses, Author (James, Joyce))

Notice that these literals, these objects, are not asserted. Objects cannot be asserted or queried (only relations).
Functors

An example assertion and an example query:

Owns (John, Book (WutheringHeights, Author(Emily, Bronte))).

Owns (John, Book (x, Author(Emily, Bronte)))?

x = WutheringHeights
Syntactically, functor symbols and relation symbols are the same. This can create great confusion to the unwary Prolog programmer.

In Visual Prolog it is possible to declare domains.

domains
 authors = author(symbol,symbol)
 books = book(symbol,authors)

And it is possible to type predicates

predicates
 owns(symbol,books)

Visual Prolog alerts the programmer to uses of functor and relation symbols inconsistent with these declarations.

These declarations are not part of standard Prolog, but no Prolog programmer should violate these distinctions.
Anatomy of a Literal

Atoms are nullary functor symbols (functor symbols with no arguments)

own (John, Book (x, Author (y, Brontë)))
Anatomy of a Clause

A clause is also called a rule. The tail of a clause consists of zero, one, or more literals. If zero literals in the tail, then the turnstile is omitted.

A query is a special case of a clause with no head.
A Prolog program consists of several clauses in order and a single query (lists of literals).
More Examples of Functors

The current focus is in on functors (functor symbols in upper case) and the construction of data.

Zero   Succ (Zero)   Succ (Succ (Zero))
Nil    Cons (x, Nil) Cons (x, Cons (y, Nil))
Leaf   Branch (Leaf, Leaf) Branch (Leaf, Branch (Leaf, Leaf))
Data Structures

But you have to be disciplined to use the free-wheeling functors; Prolog does not check for consistency, spelling, and so on.

Visual Prolog has domains (functor symbols in lower case):

domains

Haskell

nat=succ(nat);zero
data Nat=Succ Nat|Zero
list=cons(symbol,list);nil
data List a=Cons a (List a)|Nil
tree=branch(symbol,tree,tree);leaf
data Tree a=B a(Tree a)(Tree a)|L

See many-sorted algebras in a universal algebra (term algebra, free or initial algebra). See Gallier, Baader and Nipkov.
As far as data is concerned, it should come a little surprise that immutable lists are important to Prolog.
Prolog has two special functors [], . — neither are identifiers

[]        /* empty list -- a nullary functor */
.(A,[])   /* binary functor . ‘cons’ */
A.[]      /* can be written infix */

A . B . C . []
A . x . B . (y . z . []) . C . []
Since these lists are difficult to read, lists are given a special syntax.

\[
\begin{align*}
\end{align*}
\]

This notation pertains only to lists of a fixed length. A comma separates each element of the list. In Prolog the notation \([x|y]\) stands for \(x.y\), and \([A, x|y]\) stands for \(A.x.y\).
An Example With a Unary Predicate

\[ P( [A,x,y] ) . \]

\[ P( [] ) ? /* no */ \]
\[ P( [A,B,C] ) ? /* x=B, y=C */ \]
\[ P( A ) ? /* no */ \]
\[ P( [A,B] ) ? /* no */ \]
\[ P( [A,B,C,D] ) ? /* no */ \]
\[ P( [B,C,D] ) ? /* no */ \]
\[ P( [A,B,[C,D]] ) ? /* x=B, y=[C,D] */ \]
\[ P( [[A],B,C] ) ? /* no */ \]
An Example With a Binary Predicate

R ( [A,x,y], [x,B] ).

R ( [A,B], [A,B] ) ? /* no */
R ( [A], [B] ) ? /* no */
R ( [A,B,C], [A,B] ) ? /* no */
R ( [A,B,C], z ) ? /* x=B,y=C,z=[B,B] */
R ( z, [A,B] ) ? /* x=A,y=_,z=[A,A,_] */
Prolog and Lists

Q ([x|y]).

Q ([])? /* no */
Q ([A])? /* x=A, y=[] */
Q ([A,B])? /* x=A, y=[B] */
Q ([A,B,C])? /* x=A, y=[B,C] */

P ([x,A|y]).

P ([])? /* no */
P ([A])? /* no */
P ([A,B])? /* no */
P ([A,B,C])? /* no */
P ([C,A,B])? /* x=C, y=[B] */
Prolog and Lists

append ([], x) = x
append (x:xs, ys) = x:(append (xs, ys))

reverse [] = []
reverse (x:xs) = append (reverse xs, [x])
Append ([], x, x).
Append ([head|tail], x, [head|appended_tail]) :-
    Append (tail, x, appended_tail).

Reverse ([], []).
Reverse ([head|tail], result) :-
    Reverse (tail, reversed_tail),
    Append (reversed_tail, [head], result).
Consider the first of the two rules for Append

Append ([], x, x).

Append ([], [A, B, C], [A, B, D])? /* no */
Append ([], [A, B, C], [A, B, C])? /* yes */
Append ([], [A, B], z)? /* z=[A, B] */
Append ([], y, [D, E])? /* y=[D, E] */
Append (x, [F, G], [F, G])? /* x=[] */
Recall the two rules for Append

\[
\text{Append \([\[]\), \(x\), \(x\).} \\
\text{Append \([h|t]\), \(x\), \([h|l]\) :- Append \(t\), \(x\), \(l\).} \\
\]

\[
\text{Append \([A,B]\), \([C,D]\), \([H,I,J]\)?} \quad /* \text{no} */
\]
Recall the two rules for Append

\[
\text{Append } ([], x, x).
\]
\[
\text{Append } ([h|t], x, [h|l]) :- \text{ Append } (t, x, l).
\]

\[
\text{Append } ([A,B], [C,D], [A,B,C])?
\]
\[
h=A, \ t=[B], \ x=[C,D], \ l=[B,C]
\]
\[
\text{Append } ([B], [C,D], [B,C])?
\]
\[
h=B, \ t=\[], \ x=[C,D], \ l=[C]
\]
\[
\text{Append } ([\], [C,D], [C])? \quad /* \text{ no } \quad */
\]
Recall the two rules for Append

Append ([], x, x).
Append ([h|t], x, [h|l]) :- Append (t, x, l).

Append ([A,B], [C,D], [A,B,C,D])?
  h=A, t=[B], x=[C,D], l=[B,C,D]
Append ([B], [C,D], [B,C,D])?
  h=B, t=[], x=[C,D], l=[C,D]
Append ([], [C,D], [C,D])?   /* yes */
Any computable function can be computed in Prolog.

```
Plus (Zero, x, x). /* 0+x=x */
/* (x+1)+y=z+1 if x+y=z */
Plus (Succ(x), y, Succ(z)) :- Plus (x,y,z).
```

```
Mult (Zero, x, Zero). /* 0*x=0 */
/* (x+1)*y=z if x*y=w and w+y=z */
Mult (Succ(x), y, z) :- Mult (x,y,w), Plus(w,y,z).
```
Unification

Female (Alice) Female (Alice)
Female (Alice) Female (Victoria)
Female (Alice) Male (Albert)

Book (JaneEyre, Author (x, Bronte))
Book (JaneEyre, Author (Charlotte, Bronte))

Cons (x, Cons (She, Cons (It, Nil)))
Cons (He, Cons (She, Cons (y, Nil)))

Not all are so easy.

A(x,B,y) A(z,z,z)
Introduction to PROLOG Programming

(Unification)
## Unification

<table>
<thead>
<tr>
<th>term</th>
<th>term</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C(x, C(y, C(z, N)))$</td>
<td>$C(He, C(She, C(It, N)))$</td>
<td>$x=He, y=She, z=It$</td>
</tr>
<tr>
<td>$C(We, N)$</td>
<td>$C(x, y)$</td>
<td>$x=We, y=N$</td>
</tr>
<tr>
<td>$C(They, C(You, N))$</td>
<td>$C(You, C(x, N))$</td>
<td>fails</td>
</tr>
<tr>
<td>$C(x, C(She, C(It, N)))$</td>
<td>$C(He, C(She, C(y, N)))$</td>
<td>$x=He, y=It$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>term</th>
<th>term</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[x, y, z]$</td>
<td>$[He, She, It]$</td>
<td>$x=He, y=She, z=It$</td>
</tr>
<tr>
<td>$[We]$</td>
<td>$[x \mid y]$</td>
<td>$x=We, y=[]$</td>
</tr>
<tr>
<td>$[They, You]$</td>
<td>$[You, x]$</td>
<td>fails</td>
</tr>
<tr>
<td>$[x, She, It]$</td>
<td>$[He, She, y]$</td>
<td>$x=He, y=It$</td>
</tr>
</tbody>
</table>
Unification play an important role in many areas of symbolic computation:

- Logic programming (Prolog),
- Automated deduction (theorem proving),
- Constraint-based programming,
- Type inferencing (Haskell, ML),
- Knowledge-base systems; and
- Computational linguistics (features structures, unification grammars).

**Definition**

A *term* is constructed from variables and (uninterpreted) function symbols, e.g., \( a \), \( f(b) \), \( g(x, a) \).

**Definition**

A *substitution* is a finite association of terms to variables, e.g.,

\[
\sigma = \{ x_1 \mapsto t_1, \ x_2 \mapsto t_2, \ x_3 \mapsto t_3, \ldots \}
\]

The term associated by \( \sigma \) with variable \( x \) is denoted \( \sigma(x) \).
There might be no bindings in a substitution $\sigma = \{\}$; this is the empty or the identity substitution.

The value of term $t$ under the substitution $\sigma$, denoted $t\sigma$, is the term formed by simultaneously replacing all variables $v$ in $t$ by $\sigma(v)$. (We don’t bother to give a precise definition, see the examples.)

**Definition (Unification)**

Two terms $t$, and $s$, are said to unify if there is a substitution $\sigma$ such the value of the two terms are equal (symbol by symbol). $t\sigma = s\sigma$. 
Definition (Unifying Substitution)
A *unifying substitution* is a substitution that unifies two terms.

Definition (Unification Problem)
The *unification problem* is given two terms find a unifying substitution, if one exists.

Definition (Most General Substitution)
A *most general substitution* is a substitution with no irrelevant bindings.

(We don’t bother to give a precise definition.)
Substitution Examples

Knight definition 1.5, page 95.

Definition (Most General Unifier (MGU))

A unifier $\sigma$ of terms $s$ and $t$ is called a most general unifier (MGU) of $s$ and $t$ if for any other unifier $\theta$, there is a substitution $\tau$ such that $\tau \sigma = \theta$. 
Substitution Examples

Suppose $\sigma$ is $\{x_1 \mapsto y, \ x_2 \mapsto f(a), \ x_3 \mapsto f(g(b, y)), \ x_4 \mapsto f(x_2)\}$

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\sigma(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$y$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$f(a)$</td>
</tr>
<tr>
<td>$f(a)$</td>
<td>$f(a)$</td>
</tr>
<tr>
<td>$f(y)$</td>
<td>$f(y)$</td>
</tr>
<tr>
<td>$f(x_1)$</td>
<td>$f(y)$</td>
</tr>
<tr>
<td>$f(x_2)$</td>
<td>$f(f(a))$</td>
</tr>
<tr>
<td>$f(x_3)$</td>
<td>$f(f(g(b, y)))$</td>
</tr>
</tbody>
</table>
Substitution Examples

Suppose $\sigma$ is $\{x_1 \mapsto y, x_2 \mapsto f(a), x_3 \mapsto f(g(b, y)), x_4 \mapsto f(x_2)\}$

<table>
<thead>
<tr>
<th>$t$</th>
<th>$\sigma(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(x_2, x_3)$</td>
<td>$g(f(a), f(g(b, y)))$</td>
</tr>
<tr>
<td>$g(x_2, x_4)$</td>
<td>$g(f(a), f(x_2))$</td>
</tr>
<tr>
<td>$g(f(x_1), g(x_2, c))$</td>
<td>$g(f(y), g(f(a), c))$</td>
</tr>
<tr>
<td>$h(g(f(x_1), g(x_2, c)), x_4)$</td>
<td>$h(g(f(y), g(f(a), c)), f(x_2))$</td>
</tr>
<tr>
<td>$h(g(f(x_1), g(x_2, c)), f(x_3))$</td>
<td>$h(g(f(y), g(f(a), c)), f(f(g(b, y))))$</td>
</tr>
</tbody>
</table>
Examples

Find the most general unifier for the following pairs of terms (x, y, and z are variables), if it exists.

\[
\begin{align*}
  x & \quad a & \quad \{x \mapsto a\} \\
  a & \quad z & \quad \{z \mapsto a\} \\
  x & \quad f(b) & \quad \{x \mapsto f(b)\} \\
  x & \quad f(y) & \quad \{x \mapsto f(y)\} \\
  f(a) & \quad y & \quad \{y \mapsto f(a)\} \\
  g(z, c) & \quad x & \quad \{x \mapsto g(z, c)\} \\
  x & \quad x & \quad \{\} \quad \text{the empty or id subst} \\
  x & \quad y & \quad \{x \mapsto y\} \text{ or } \{y \mapsto x\} \\
  g(a, b) & \quad g(a, b) & \quad \{\} \quad \text{the empty or id subst} \\
  g(a, x) & \quad g(b, x) & \quad \text{no unifer} \quad a \neq b \\
  g(a, x) & \quad h(a, b) & \quad \text{no unifer} \quad g \neq h
\end{align*}
\]
Examples

Find the most general unifier for the following pairs of terms (\(x\), \(y\), and \(z\) are variables), if it exists.

\[
\begin{align*}
g(x, g(y, z)) & \quad g(a, g(f(b), c)) & \{x \mapsto a, \, y \mapsto f(b), \, z \mapsto c\} \\
g(x, a) & \quad g(c, y) & \{x \mapsto c, \, y \mapsto a\} \\
f(g(c, x)) & \quad f(g(y, d)) & \{x \mapsto d, \, y \mapsto c\} \\
g(f(x), h(y, b)) & \quad g(z, x) & \{z \mapsto f(h(y, b)), \, x \mapsto h(y, b)\} \\
j(x, b, y) & \quad j(y, b, c) & \{x \mapsto c, \, y \mapsto c\} \\
j(x, b, c) & \quad j(a, b, x) & \text{no unifier}
\end{align*}
\]
We next present a unification algorithm by example. This algorithm scans from left to right building the substitution up one binding at a time. Terms are matched symbol by symbol. Variables match anything and give rise to new bindings; if two terms can’t be unified, then the algorithm terminates in failure. The bindings must be properly substituted as we go along, and the final substitution is obtained by properly substituting the bindings in all the previous bindings.
### Unification

<table>
<thead>
<tr>
<th>$s(x, g(f(z), v, a))$</th>
<th>$s(f(y), g(x, h(x), y))$</th>
<th>$s$ matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s(x, g(f(z), v, a))$</td>
<td>$s(f(y), g(x, h(x), y))$</td>
<td>$x \mapsto f(y)$</td>
</tr>
<tr>
<td>$s(f(y), g(f(z), v, a))$</td>
<td>$s(f(y), g(f(y), h(f(y)), y))$</td>
<td>substitute</td>
</tr>
<tr>
<td>$s(f(y), g(f(z), v, a))$</td>
<td>$s(f(y), g(f(y), h(f(y)), y))$</td>
<td>$g$ matches</td>
</tr>
<tr>
<td>$s(f(y), g(f(z), v, a))$</td>
<td>$s(f(y), g(f(y), h(f(y)), y))$</td>
<td>$f$ matches</td>
</tr>
<tr>
<td>$s(f(y), g(f(z), v, a))$</td>
<td>$s(f(y), g(f(y), h(f(y)), y))$</td>
<td>$z \mapsto y$</td>
</tr>
<tr>
<td>$s(f(y), g(f(y), v, a))$</td>
<td>$s(f(y), g(f(y), h(f(y)), y))$</td>
<td>substitute</td>
</tr>
<tr>
<td>$s(f(y), g(f(y), v, a))$</td>
<td>$s(f(y), g(f(y), h(f(y)), y))$</td>
<td>$v \mapsto h(f(y))$</td>
</tr>
</tbody>
</table>
Unification

\[
\begin{align*}
  s(x, g(f(z), v, a)) & \quad s(f(y), g(x, h(x), y)) & \quad s \text{ matches} \\
  s(x, g(f(z), v, a)) & \quad s(f(y), g(x, h(x), y)) \quad x \mapsto f(y) \\
  s(f(y), g(f(z), v, a)) & \quad s(f(y), g(f(y), h(f(y)), y)) \quad g \text{ matches} \\
  s(f(y), g(f(z), v, a)) & \quad s(f(y), g(f(y), h(f(y)), y)) \quad f \text{ matches} \\
  s(f(y), g(f(z), v, a)) & \quad s(f(y), g(f(y), h(f(y)), y)) \quad z \mapsto y \\
  s(f(y), g(f(y), v, a)) & \quad s(f(y), g(f(y), h(f(y)), y)) \quad v \mapsto h(f(y)) \\
  s(f(y), g(f(y), h(f(y)), a)) & \quad s(f(y), g(f(y), h(f(y)), y)) \quad \text{substitute} \\
  s(f(y), g(f(y), h(f(y)), a)) & \quad s(f(y), g(f(y), h(f(y)), y)) \quad y \mapsto a \\
  s(f(a), g(f(a), h(f(a)), a)) & \quad s(f(a), g(f(a), h(f(a)), a)) \quad \text{substitute}
\end{align*}
\]

Resulting in the (most general) unifying substitution:

\[
\{x \mapsto f(a), z \mapsto a, v \mapsto h(f(a)), y \mapsto a\}
\]
Occurs Check

Suppose $\sigma$ is $\{\langle x \mapsto g(b, x) \rangle\}$

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>t$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>f(y)</td>
<td>f(y)</td>
<td></td>
</tr>
<tr>
<td>f(x)</td>
<td>f(g(b, x))</td>
<td></td>
</tr>
<tr>
<td>f(g(b, x))</td>
<td>f(g(b, g(b, x)))</td>
<td></td>
</tr>
</tbody>
</table>

The binding $\langle x \mapsto g(b, x) \rangle$ can never contribute to making a term containing $x$ unify with another term containing $x$.

*Standard Prolog omits the occurs check during unification!* ‘[In 1973] the “occur check” disappeared as it was found to be too costly.’ Colmerauer and Roussel, November 1992, “The Birth of Prolog” in *History of Programming Languages – II.*
Implementations offering sound unification for all unifications are Qu-Prolog and Strawberry Prolog and (optionally, via a runtime flag): XSB and SWI-Prolog.
Unification Algorithm

Initialize $\theta$ to be empty
Clear the stack; push $t = s$ on the stack

while stack not empty do
  pop $X = Y$ from the stack
  case
  X and Y the same variable:
    continue
  X is a variable not occurring in Y:
    substitute Y for X in stack and $\theta$
    add $X \mapsto Y$ to $\theta$
  Y is a variable not occurring in X:
    substitute X for Y in stack and $\theta$
    add $Y \mapsto X$ to $\theta$
  $X = f(X_1, \ldots, X_n)$ and $Y = f(Y_1, \ldots, Y_n)$:
    push $X_i = Y_i$ on the stack
  otherwise, fail.

Introduction to PROLOG Programming (Unification)
function UNIFY(t1, t2) \(\implies\) (unifiable: Boolean, \(\sigma\): substitution)
\(\begin{align*}
&\text{begin} \\
&\text{pairs-to-unify} \leftarrow \{(t1, t2)\} \\
&\text{for each node } z \text{ in } t1 \text{ and } t2, \\
&\hspace{1em} z\text{.class} \leftarrow z \\
&\text{while } \text{pairs-to-unify} \neq \emptyset \text{ do} \\
&\hspace{1em}\text{begin} \\
&\hspace{2em}(x, y) \leftarrow \text{pop (pairs-to-unify)} \\
&\hspace{2em}u \leftarrow \text{FIND}(x) \\
&\hspace{2em}u \leftarrow \text{FIND}(y) \\
&\hspace{2em}\text{if } u \neq v \text{ then} \\
&\hspace{3em}\text{if } u \text{ and } v \text{ are not variables, and } \text{u.symbol} \neq \text{v.symbol} \text{ or} \\
&\hspace{4em}\text{numnode}(u) \neq \text{numnode}(v) \text{ then} \\
&\hspace{5em}\text{return } (\text{false, nil}) \\
&\hspace{2em}\text{else} \\
&\hspace{3em}\text{begin} \\
&\hspace{4em}w \leftarrow \text{UNION}(u, v) \\
&\hspace{4em}\text{if } w = v \text{ and } u \text{ is a variable} \text{ then} \\
&\hspace{5em}u\text{.class} \leftarrow v \\
&\hspace{4em}\text{if neither } v \text{ nor } u \text{ is a variable} \text{ then} \\
&\hspace{5em}\text{begin} \\
&\hspace{6em}\text{let } (u_1, \ldots, u_n) \text{ be } u\text{.subnodes} \\
&\hspace{6em}\text{let } (v_1, \ldots, v_n) \text{ be } v\text{.subnodes} \\
&\hspace{6em}\text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
&\hspace{7em}\text{push } ((u_i, v_i), \text{pairs-to-unify}) \\
&\hspace{5em}\text{end} \\
&\hspace{4em}\text{end} \\
&\hspace{2em}\text{end} \\
&\text{end} \\
&\text{Form a new graph composed of the root nodes of the equivalence classes.} \\
&\text{This graph is the result of the unification.} \\
&\text{If the graph has a cycle, return } (\text{false, nil}), \text{ but the terms are } \text{ininitely unifiable.} \\
&\text{If the graph is acyclic, return } (\text{true, } \sigma), \text{ where } \sigma \text{ is a substitution in which any variable } x \text{ is mapped on to the} \\
&\text{root of its equivalence class, that is, FIND}(x). \\
&\text{end}
\end{align*}\)

Figure 4. A version of Huet’s unification algorithm.


Unification Algorithm

- linear time $O(n)$

- near linear time $O(nA^{-1}(n))$, union-find
Prolog and Logic

Mortal(x) :- Man(x). /* all men are mortal */
Man(Socrates). /* Socrates is a man */
Mortal(Socrates)? /* Is Socrates mortal? */

Assuming $\forall x \ M(x) \Rightarrow R(x)$ and $M(s)$, the laws of logic entail that $R(s)$.

If Prolog finds a solution, then the query (usually) logically follows from the assumptions.
Prolog and Logic

Prolog is not a satisfactory theorem prover for two reasons: DFS and omitting the occurs check.

\[ P :- P. \] /* of course, but useless */

\[ P. \] /* assert P */

Does \( P \) logically follow from \( P \) (and \( P \Rightarrow P \))? Yes, but the first rule throws Prolog into an infinite loop. Solved by BFS.
Knight, Figure 6, page 201.

/* Any number is less than its successor. */
Less(y,Succ(y)). /* for all y, y < y+1 */

/* Is there a number whose successor is less than it? */
Less(Succ(x),x)? /* does some x, x+1<x */

/* Everybody has a friend they like. */
Likes(y,Friend(y)).
/* Does everybody like themselves? */
Likes(x,x)?

Prolog finds a solution, but goes into an infinite loop printing the binding for x. The absence of the occurs-check renders Prolog unsound.
NB. You cannot express negation in Prolog, but we do not blame Prolog for being unable to prove things it cannot express.
* Clauses

**Clause.** A clause is a logical formula written in a special way using two sets of literals:

\[ P_1, \ldots, P_n \leftarrow Q_1, \ldots, Q_k \]

In more ordinary logical notation, a clause means:

\[(Q_1 & \cdots & Q_k) \Rightarrow (P_1 | \cdots | P_n)\]

(Universal quantification is implicit, existential quantification through skolemization.)
* Clausal Form

All formulas can be put in clausal form. For example,

\[
\begin{align*}
\bot & \quad \bot \iff \top \\
A \mid B & \quad A, B \iff \top \\
A \mid \neg B & \quad A \iff B \\
\neg A & \quad \bot \iff A \\
\neg (A \& B) & \quad \bot \iff A, B \\
A \& B & \quad A \iff \top \\
B & \iff \top \\
(A \& B) \Rightarrow (C \mid D) & \quad C, D \iff A, B \\
\neg A \mid \neg B \mid C \mid D & \quad C, D \iff A, B
\end{align*}
\]
Resolution

Robinson’s resolution principle:

\[
\frac{Q_1, \ldots, Q_n \leftarrow P_1, P_2, \ldots, P_k \hspace{1cm} R_1, R_2, \ldots, R_m \leftarrow S_1, \ldots, S_l}{(Q_1, \ldots, Q_n, R_2, \ldots, R_m)\sigma \leftarrow (P_2, \ldots, P_k, S_1, \ldots, S_n)\sigma}
\]

where \(P_1\) and \(R_1\) unify under the most general unifying substitution \(\sigma\).

One rule of inference for first-order predicate logic.

Horn Clauses

Horn clause. Special case of a clause where \( n \) is 0 or 1. A Prolog rule or fact has \( n = 1 \).

\[
P_1 \leftarrow Q_1, \ldots, Q_k
\]

A Prolog query has \( n = 0 \)

\[
\bot \leftarrow Q_1, \ldots, Q_k
\]

Horn clause resolution principle:

\[
\frac{\leftarrow Q_1, Q_2, \ldots, Q_n}{\leftarrow (P_1, \ldots, P_k, Q_2, \ldots, Q_n)\sigma}
\]

\[
P_0 \leftarrow P_1, P_2, \ldots, P_k
\]

where \( Q_1 \) and \( P_0 \) unify under the most general unifying substitution \( \sigma \).

Resolution principle in other notation:

\[
\neg (P_1 \& \ldots \& P_k \& Q_2 \& \ldots \& Q_n)\sigma
\]

\[
\frac{\neg (Q_1 \& \ldots \& Q_n)}{P_1 \& \ldots \& P_k \Rightarrow P_0}
\]
\[
\bot \iff R, Q_1, \ldots, Q_n \quad R \iff P_1, \ldots, P_k
\]
\[
\bot \iff P_1, \ldots, P_k, Q_1, \ldots, Q_n
\]

\[
R, Q_1, \ldots, Q_n \Rightarrow \bot \quad P_1, \ldots, P_k \Rightarrow R
\]
\[
P_1, \ldots, P_k, Q_1, \ldots, Q_n \Rightarrow \bot
\]

\[
\neg(Q_1, \ldots, Q_n) \lor \neg R \quad \neg(P_1, \ldots, P_k) \lor R
\]
\[
\neg(P_1, \ldots, P_k, Q_1, \ldots, Q_n)
\]
SLD resolution (Selective Linear Definite clause resolution) is the basic inference rule used in logic programming. It is a refinement of resolution, which is both sound and refutation complete for Horn clauses.

The name "SLD resolution" was given by Maarten van Emden for the unnamed inference rule introduced by Robert Kowalski.

SLD resolution is non-deterministic in the sense that it does not determine the search strategy for exploring the search tree. Prolog searches the tree depth-first, one branch at a time, using backtracking when it encounters a failure node.

Linear Input Resolution

Nilson, AI, section 5.2.4.
* Linear Input Resolution

\[ \leftarrow G, G, \ldots, G \quad P \leftarrow Q, \ldots, Q \]

\[ \leftarrow G, G, \ldots, G \quad P \leftarrow Q, \ldots, Q \]

\[ \leftarrow G, G, \ldots, G \quad P \leftarrow Q, \ldots, Q \]

\[ \leftarrow \quad P \leftarrow Q, \ldots, Q \]
Negation as failure

\[
\text{not}(C) \leftarrow \text{call}(C),!,\text{fail}.
\]
\[
\text{not}(C).
\]

Not logically correct

\[
P(a).
\]
\[
Q(x) \leftarrow \text{not} \ (P(b)).
\]
\[
Q(b)? \quad /* \text{succeeds because} \ P(b) \ \text{does not} */
\]

But \( P(a) \) and \( \forall x \neg P(b) \Rightarrow Q(x) \) does not imply \( Q(b) \). The absence of the assertion \( P(b) \) does not entail \( \neg P(b) \).

\textit{Closed-world assumption}. All facts are known/asserted, so any missing assumption must necessarily be false. E.g., \( \neg P(b) \) must be true, ow \( P(b) \) would have been asserted.