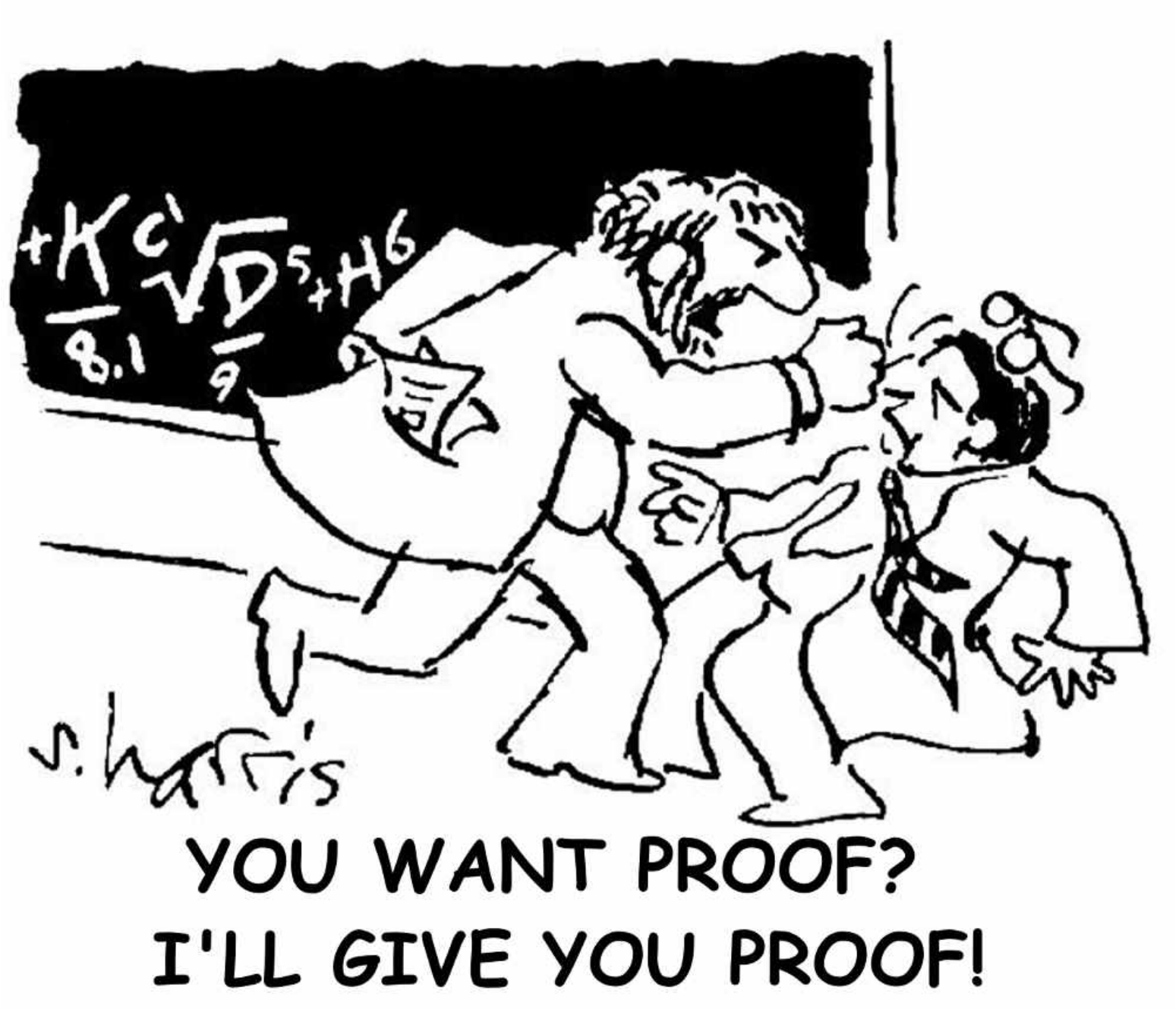

+K c' √D5+H6
8.1 9
S. Harris
YOU WANT PROOF?
I'LL GIVE YOU PROOF!

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.

Prolog

Logic is not the route to sleep.

Garrison Keillor, A Prairie Home Companion, 5 January 1985.

Prolog Overview

- Introduction: goals, definitions, history
- Prolog programming: facts, queries, variables, rules, recursion
- Backward chaining (resolution)
- $*$ Cut
- 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

- Declarative programming – what versus how.

To a small degree the SETL language exhibits declarative programming (cf. example of topological sort). SETL developed by J. Schwartz (NYU) around 1974.

```
$ 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 declaractive, but imperative) language Python using (not imperative, but functional) mechanism of list comprehension.

```
def range (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 range(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
```

Like Prolog the hold SETL has on being declarative is very weak.

```
(while exists n in nodes | n notin range G)
```

```
ns = [n for n in nodes if n not in range(g)]
```

The philosphical 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 search of the list.

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*.

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 Univeristy 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 misleading 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

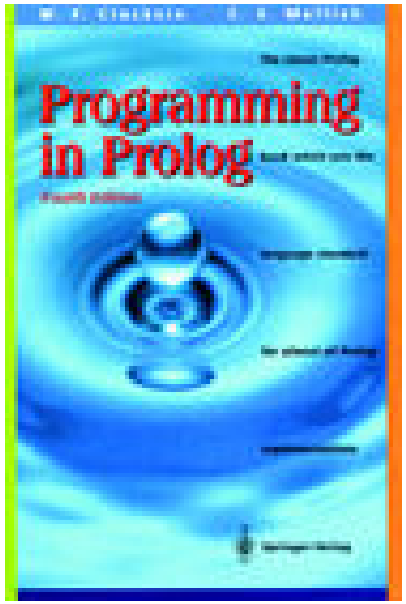
Alain Colmerauer: 1941-2017

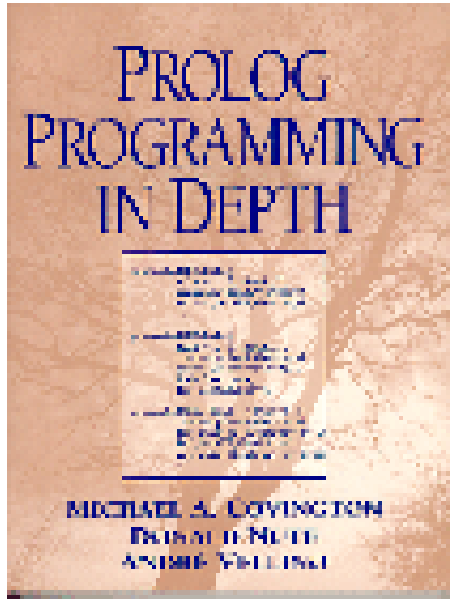
> *He was the center of a research group that, in the early 1970s, first conceived of Prolog (an abbreviation for "programmation en logique," which is French for "programming in logic"), a general-purpose logic programming language rooted in first-order logic that is well-suited for tasks that benefit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

`http://prolog-heritage.org/en/ph12.html`
`https://cacm.acm.org/news/217533-in-memoriam-alain-colmerauer-1941-2017/fulltext`

Information about Prolog

Clocksin, William F., and Christopher S. Mellish. *Programming in Prolog*, fourth edition. Springer-Verlag, Berlin, 1994. ISBN 0-387-58350-5.





Convington, Michael A., Donald Nute, André Vellino. *Prolog Programming in Depth.* Prentice-Hall, Upper Saddle River, New Jersey, 1997. ISBN 0-13-138645-X.

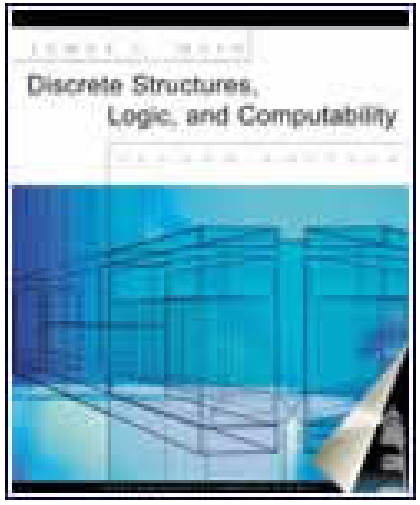
Information about Prolog

Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2001.
Colmerauer and Roussel. "The Birth of Prolog," in *History of Programming Languages* edited by Bergin and Gibson. ACM Press, 1996, pages 331–367.
Steling and Shaprio. *The Art of Prolog*. MIT Press, 1994.
O'Keefe. *The Craft of Prolog*. MIT Press, 1990.
Apt. "The Logic Programming Paradigm and Prolog," in *Concepts of Programming Languages*.
Stansifer in *The Study of Programming Languages*.

Logic

What is predicate logic?

Hein, James. *Discrete Structures, Logic, and Computability.* Jones and Bartlett, Sudbury, Massachusetts, 2002. ISBN 0-7637-1843-2. Chapter 6: Elementary Logic, Chapter 7: Predicate Logic, Chapter 8: Applied Logic, Chapter 9: Computational Logic.



Recap of First-Order Logic

$\bot$	false
$\top$	true
$A \& B$	A and B
$A \mid B$	A or B
$\neg A$	not A
$A \Rightarrow B$	A implies B
$\forall x\, P(x)$	for all x, $P(x)$
$\exists x\, P(x)$	there exists x, $P(x)$

Modus ponens, one of the classic laws of deduction

$$\frac{A \Rightarrow B \quad A}{B}$$

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

$p \Rightarrow q$: 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 $p \Rightarrow q$?

$p\&\neg q$: 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):

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.

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.

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.

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

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 Rochell 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 Rochell did not get the supervisor's position or she did not work hard.*

$s \& w \Rightarrow r$

$r \Rightarrow c$

$\neg c$

$\therefore \neg s \mid \neg w$

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.

$r \Rightarrow h$
$d \Rightarrow c$
$h \mid c \Rightarrow v$
$\neg v$
$\therefore \neg r \& \neg d$

Describe in words the rules of inference:

1. *Modus Pones*
2. *Modus Tollendo Tollens*
3. DeMorgan's laws (equivalences)
4. Conjunction introduction
5. Conjunction elimination

Use them to prove

$r \Rightarrow h$

$d \Rightarrow c$

$h \mid c \Rightarrow v$

$\neg v$

$\therefore \neg r \& \neg d$

Logic Deamon (logic.tamu.edu)

```
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

Prolog Programming

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.                          */
```

Syntax of a literal: `PredicateSymbol ( Atom ).` A fact ends in a period. 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

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, retreiver)
W = puppy(terry, frisky, poodle) ;

No
3 ?-
```

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
```

Quantifiers

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.

Variables

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 if 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.  */
```

Quantifiers

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

```
Mortal (x) :- Man (x).   /*  All men are mortal.   */
Man (Socrates).          /*  Socrates is a man.    */
Mortal (Socrates)?
yes
```

Quantification

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.

Quantification

```
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
```

Quantification

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)`

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`

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)`

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`

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)`

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)`

Example

`In (Austin, NA)?`

The search process takes the following steps:

- Goal: `In (Austin, NA)`
  Matches rule (8): `In(x,NA)` if `x=Austin`
- Goal: `In (Austin, USA)`

Example

`In (Austin, NA)?`

The search process takes the following steps:

- Goal: `In (Austin, NA)`
  Matches rule (8): `In(x,NA)` if `x=Austin`
- Goal: `In (Austin, USA)`
  Matches rule (5): `In(x,USA)` if `x=Austin`
- Goal: `In (Austin, Georgia)`

Example

`In (Austin, NA)?`

The search process takes the following steps:

- Goal: `In (Austin, NA)`
  Matches rule (8): `In(x,NA)` if `x=Austin`
- Goal: `In (Austin, USA)`
  Matches rule (5): `In(x,USA)` if `x=Austin`
- Goal: `In (Austin, Georgia)`
  No rule matches! Backtrack.
- Goal: `In (Austin, USA)`

Example

`In (Austin, NA)?`

The search process takes the following steps:

- Goal: `In (Austin, NA)`
  Matches rule (8): `In(x,NA)` if `x=Austin`
- Goal: `In (Austin, USA)`
  Matches rule (5): `In(x,USA)` if `x=Austin`
- Goal: `In (Austin, Georgia)`
  No rule matches! Backtrack.
- Goal: `In (Austin, USA)`
  Matches rule (6): `In(x,USA)` if `x=Austin`
- Goal: `In (Austin, Texas)`

Example

`In (Austin, NA)?`

The search process takes the following steps:

- Goal: `In (Austin, NA)`
  Matches rule (8): `In(x,NA)` if `x=Austin`
- Goal: `In (Austin, USA)`
  Matches rule (5): `In(x,USA)` if `x=Austin`
- Goal: `In (Austin, Georgia)`
  No rule matches! Backtrack.
- Goal: `In (Austin, USA)`
  Matches rule (6): `In(x,USA)` if `x=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

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)?`

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,` x_1`)`
  Matches rule (7): `Sister(`x_2`,y)` if x_2=`Alice`, and x_1=`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`=x_1=`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=Alice`.

- Goal: `Parents(y,Victoria,Albert)`
  Matches fact (6): `Parents (Alice, Victoria, Albert)` if `y=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/=y
```

But it is not at all clear what `x/=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 *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 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!

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 build the (imaginary) search space is immaterial. The method of traversing the search space is crucial.

Prolog does not first create the search space and then traverse it. But one cannot understand Prolog theoretically without being able to visualize the search space of a Prolog program.

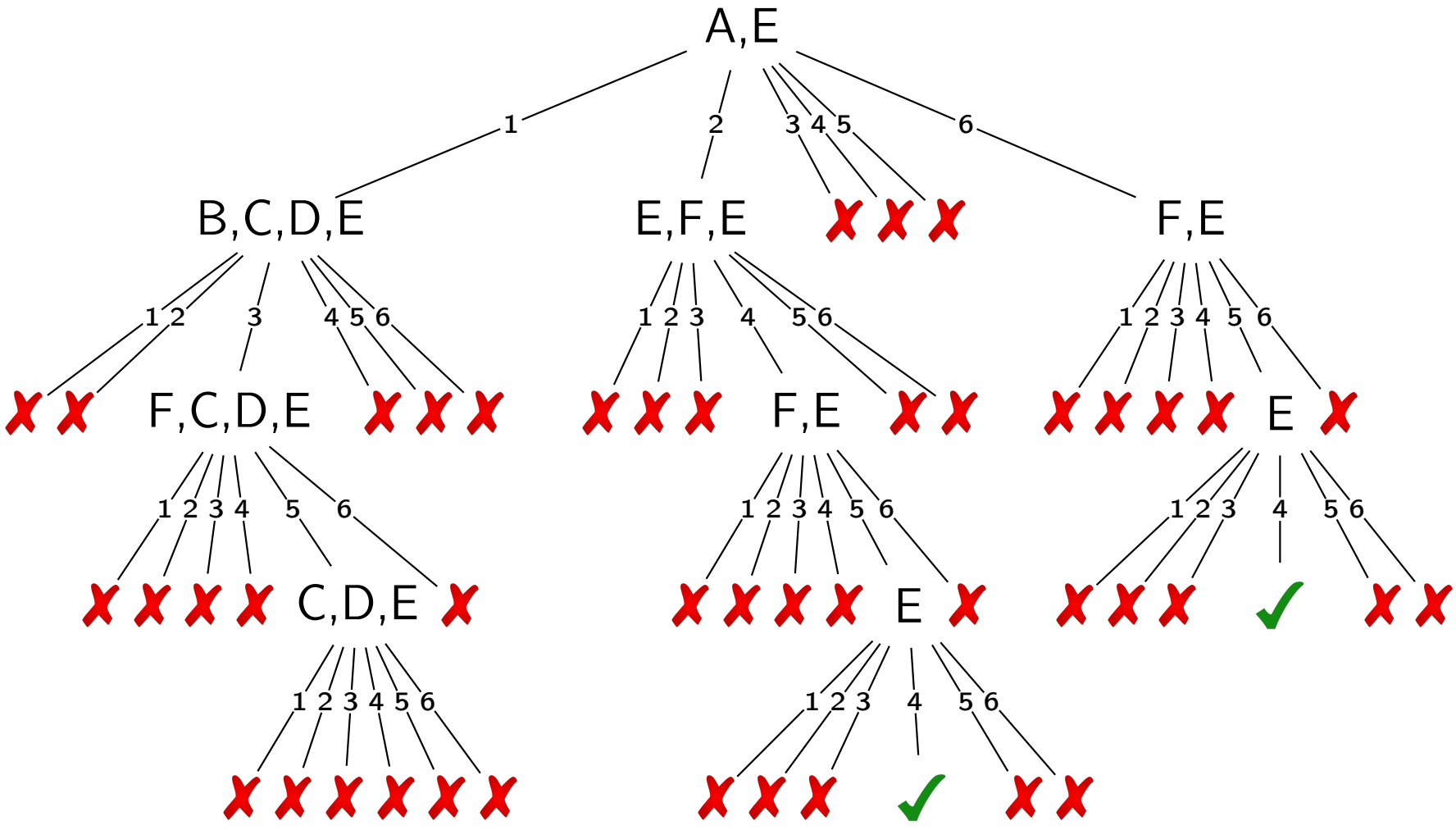
A nullary predicate is a predicate with zero arguments,
e.g., `Iam` versus `Gives(x,y,z)`.

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.
```


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

You can take shortcuts on the exam, but be careful. It may be clear to you what you mean, but if it is not perfectly clear to the grader you may lose points.

Variation I

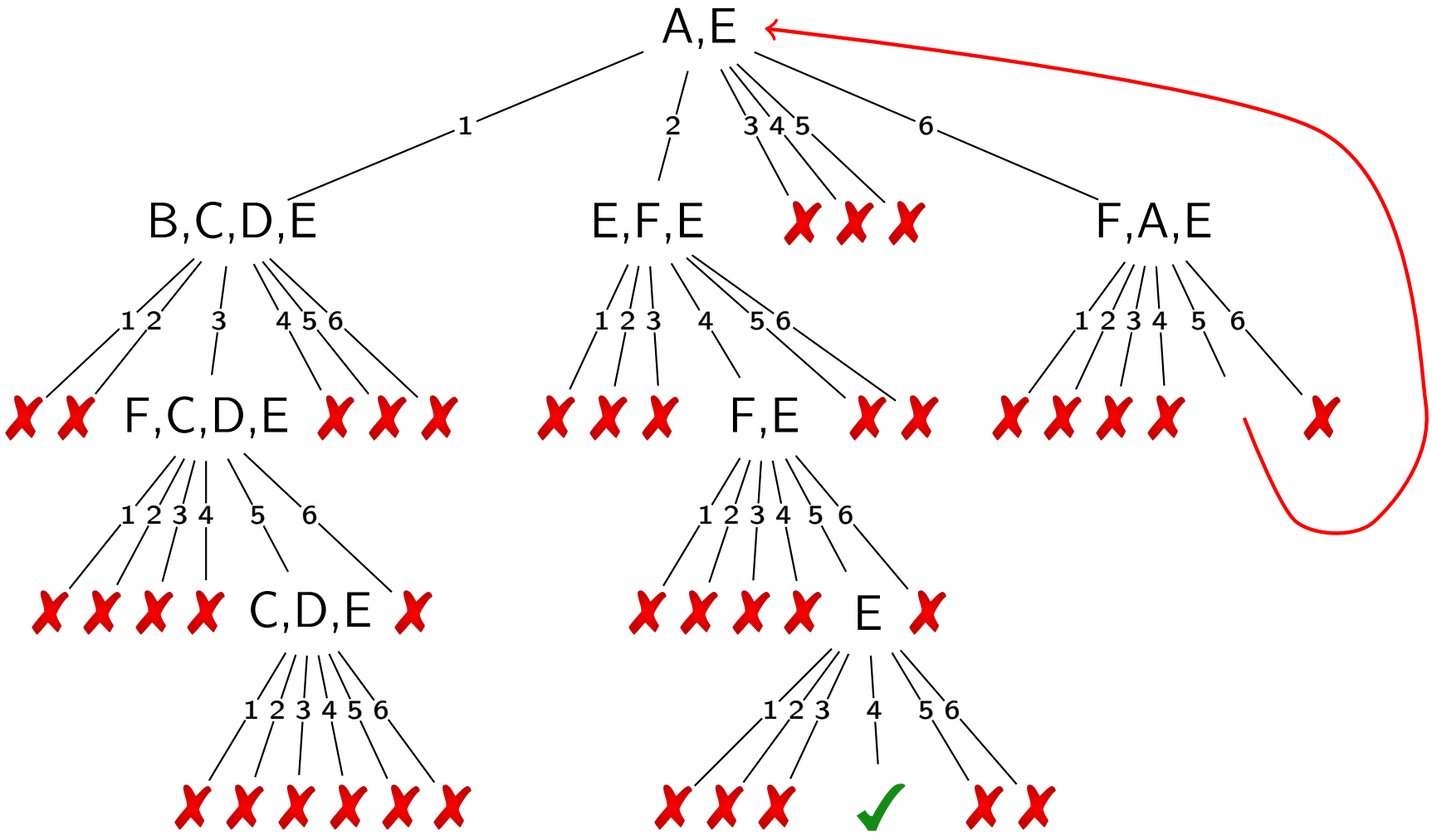
Consider the query `A,E?` in a variation of the original database of facts and rules.

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

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

Search Space (Variation I)

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).


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

Variation II

Modify the previous database of facts and rules by switching rules 1 and 6, consider the query `A,E?`.

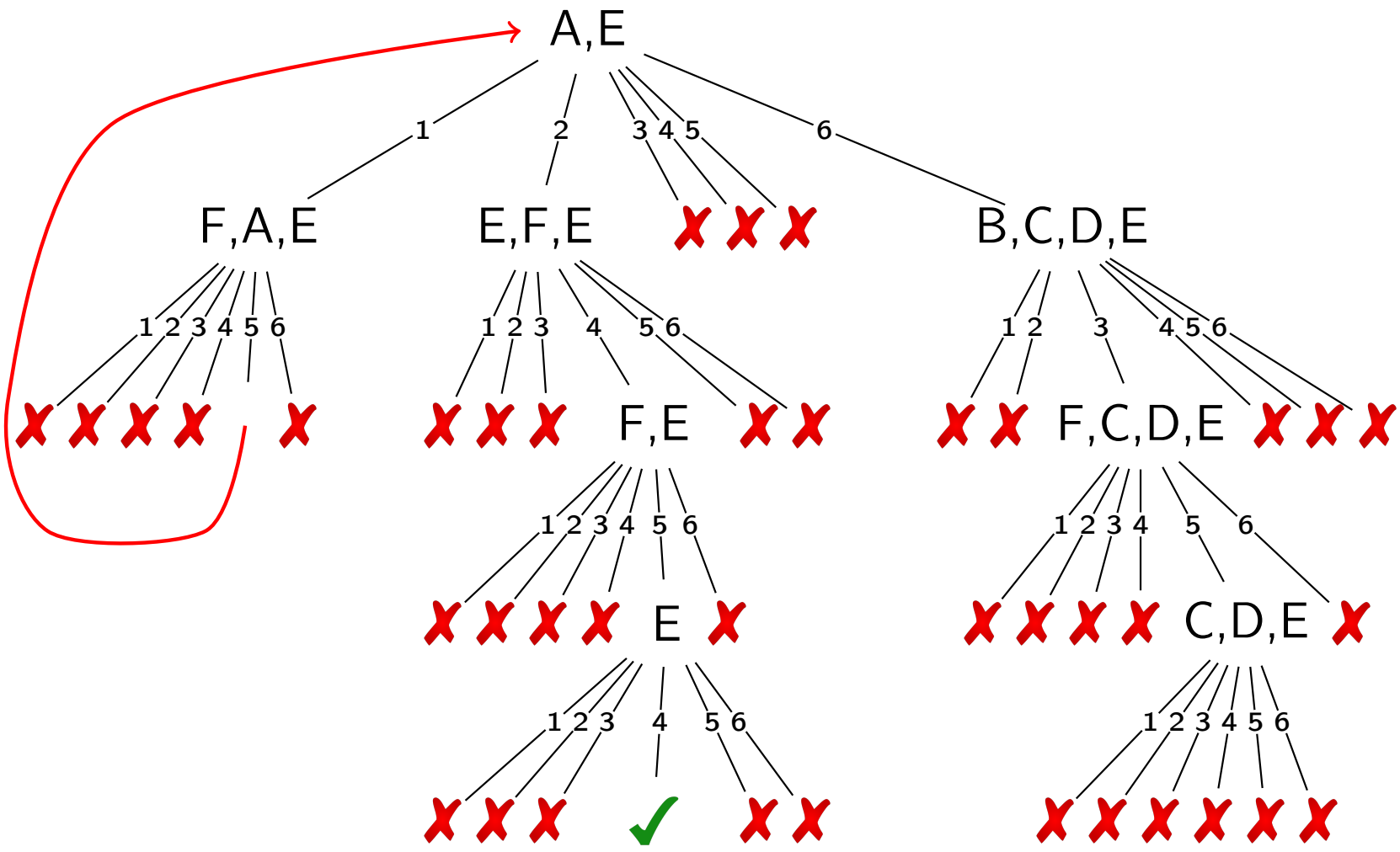
```
1   A :- B,C,D.
2   A :- E,F.
3   B :- F.
4   E.
5   F.
6   A :- F,A.
```

Variation II

Modify the previous database of facts and rules by switching rules 1 and 6, consider the query `A,E?`.

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

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

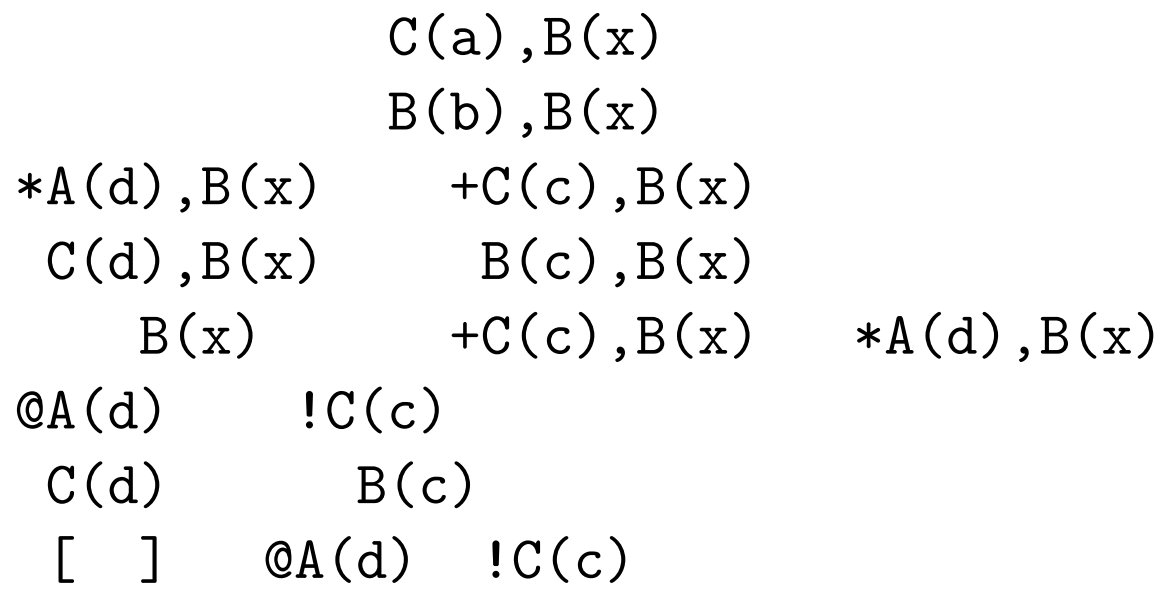

A,E
1
2
3 4 5
6
F,A,E
E,F,E
B,C,D,E
1 2 3 4 5 6
1 2 3
4
5 6
1 2
3
4 5 6
F,E
F,C,D,E
1 2 3 4 5 6
1 2 3 4
5
6
E
C,D,E
1 2 3
4
5 6
1 2 3 4 5 6

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



∗ 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.
```

* Various Built-ins

ISO Prolog standard: ISO/IEC 13211-1 was published in 1995.
It describes 102 built-ins. ISO built-ins
`asserta/1`, `assertz/1`, `call/1`, `!/0` (cut), `get_char/1`,
`unify_with_occurs_check/2`, `var/1`, etc.

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!

Monty Python

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)?
```

Monty Python

The argument is valid.

```
Witch (Girl)?
Burns (Girl)?
MadeOfWood (Girl)?
Floats (Girl)?
SameWeight(Girl,Duck)?
[+]
```

Overview

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 the 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.
First we generalize objects in Prolog to include structured objects called functors and consider their significance. Then we introduce the unification problem and unification algorithms.

Functors

Data structures are achieved in Prolog using *functors* — uninterpreted function symbols 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 are not asserted. Objects cannot be asserted or queried (only relations).
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 programmer.
In Turbo Prolog it is possible to declare domains.

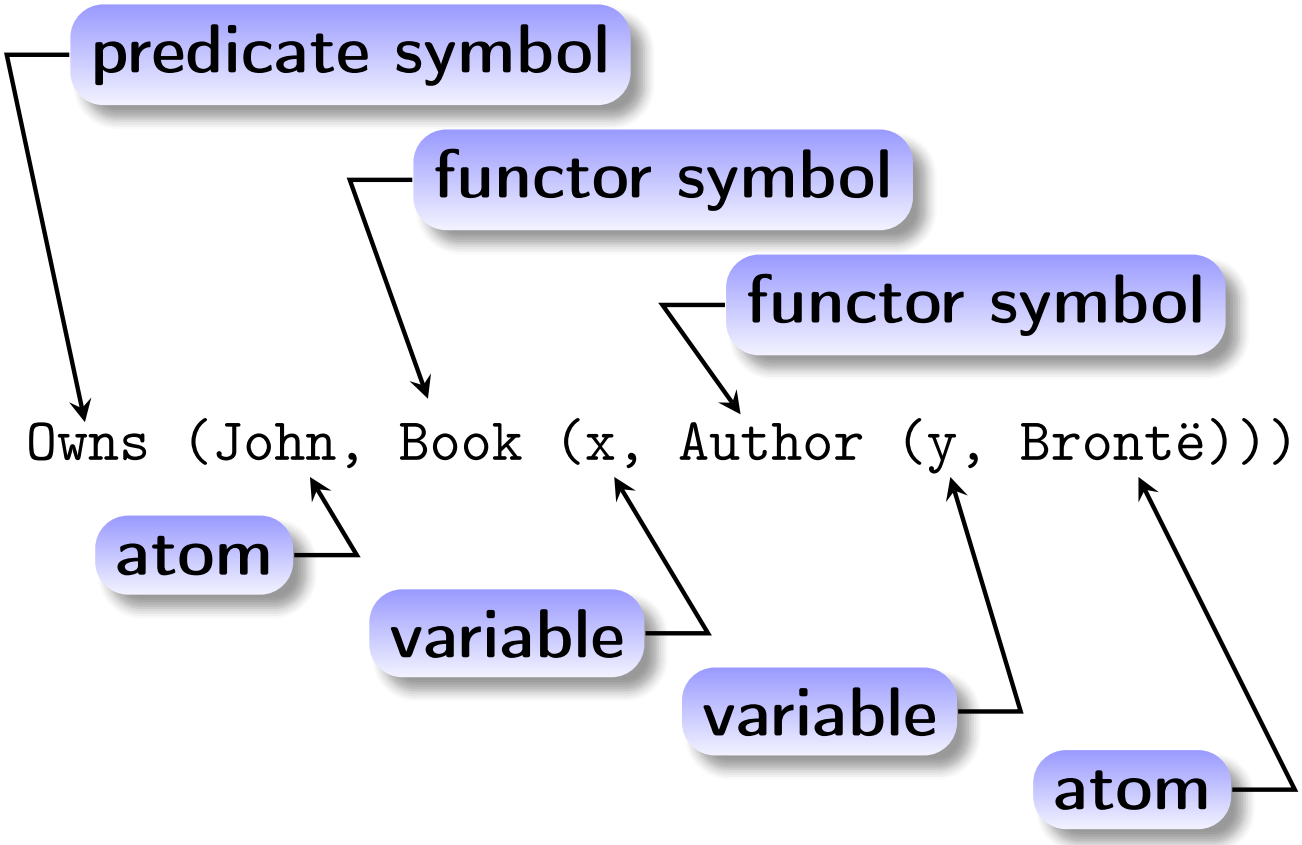
```
domains
    authors = author(symbol,symbol)
    books = book(symbol,authors)
```

And it is possible to type predicates

```
predicates
    owns(symbol,books)
```

Turbo 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



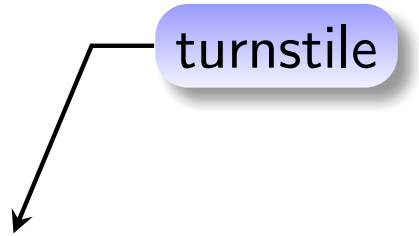
Atoms are nullary functor symbols
(functor symbols with no arguments)

Anatomy of a Clause

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

A clause is also called a rule.
The tail of a clause consists of zero, one, or more literals. If zero literals, then the turnstile is omitted.
A query is a special case of a clause with no head.

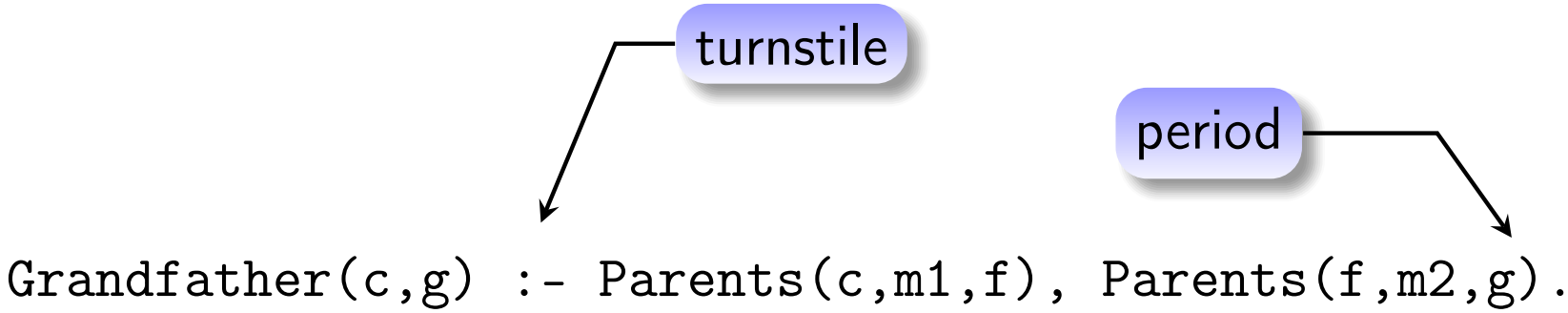
Anatomy of a Clause




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

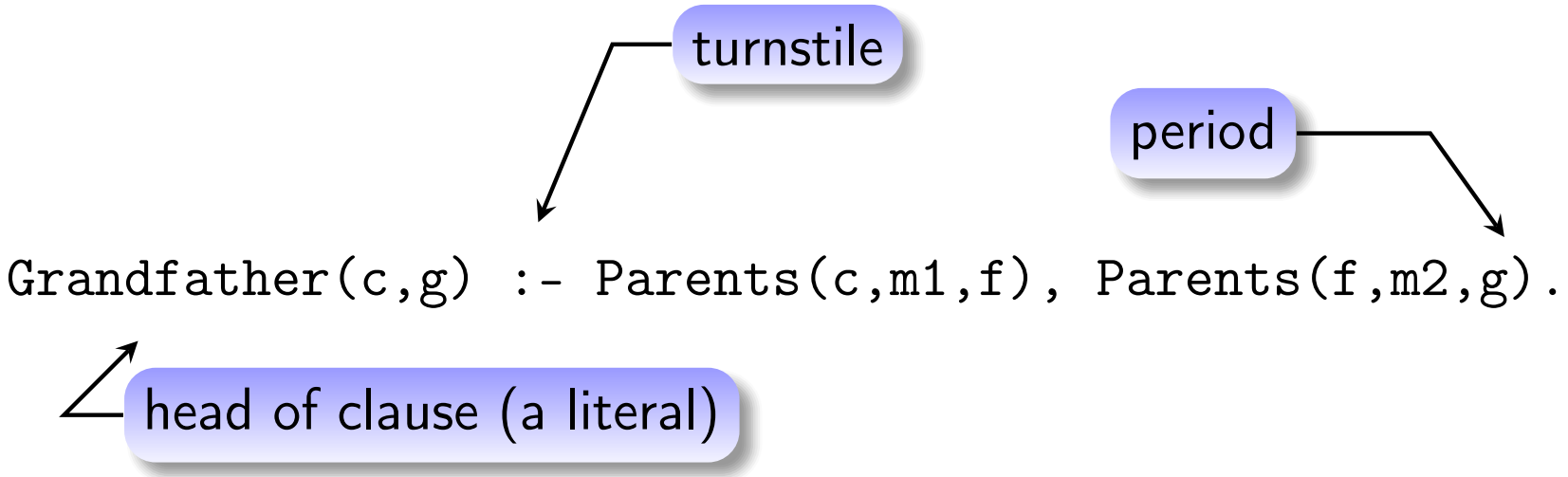
A clause is also called a rule.
The tail of a clause consists of zero, one, or more literals. If zero literals, then the turnstile is omitted.
A query is a special case of a clause with no head.

Anatomy of a Clause



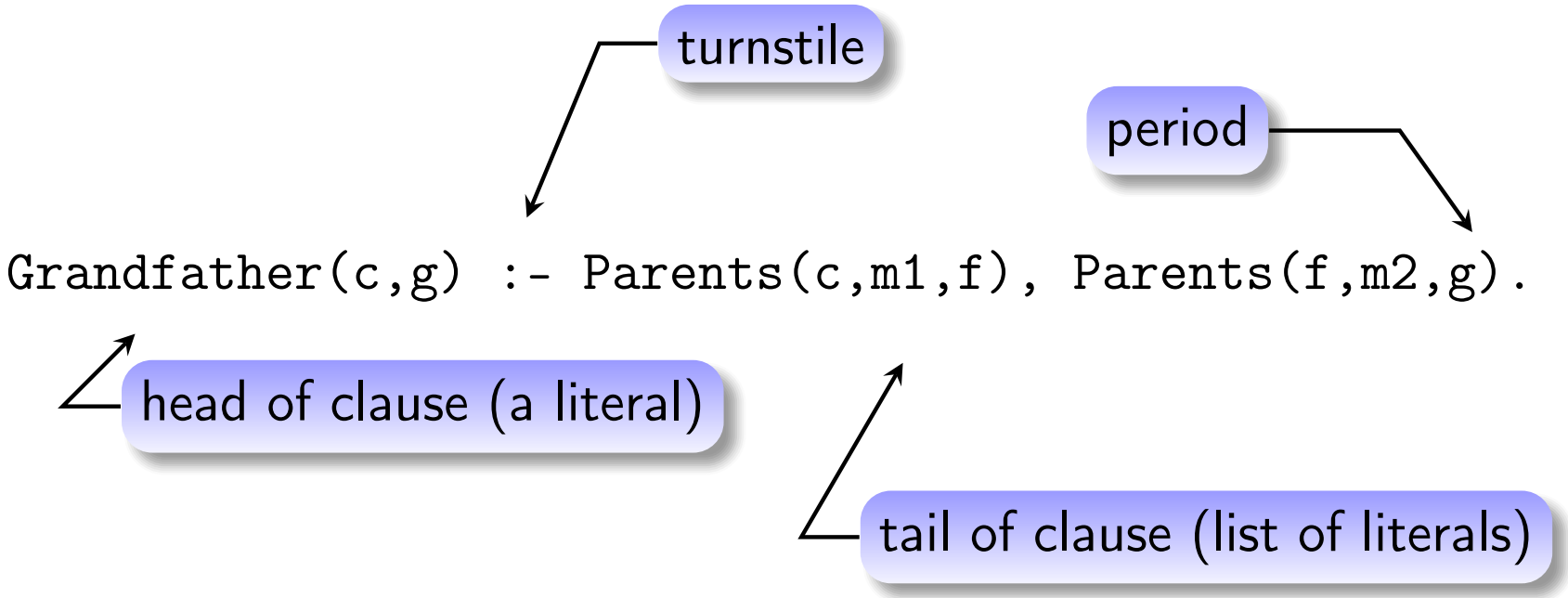
A clause is also called a rule.
The tail of a clause consists of zero, one, or more literals. If zero literals, then the turnstile is omitted.
A query is a special case of a clause with no head.

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, then the turnstile is omitted.
A query is a special case of a clause with no head.

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, then the turnstile is omitted.
A query is a special case of a clause with no head.

Prolog and Lists

Prolog has two special functors `[]`, `.`

```
[]      /* 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.

```
[A,B,C]
[A,x,B,[y,z],C]
```

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`.

Prolog and Lists

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

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

```
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,A,A], [A,B] ) ? /* x=A,y=A                     */
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] */
```

Functions

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

Functions *are* a kind of relation!

Prolog and Lists

```
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 (reverse_tail, [head], result).
```

Prolog and Lists

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=[]    */
```

Prolog and Lists

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

Append ([A,B], [C,D], [H,I,J])?   /* no */

Append ([A,B], [C,D], [A,B,C])?
h=A, t=[B], x=[C,D], l=[B,C]
Append ([B], [C,D], [B,C])?
h=B, t=[], x=[C,D], l=[C]
Append ([], [C,D], [C])?    /* no  */

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 */
```

Turing Complete

Any computable function can be computed in Prolog.
First we need that natural numbers as a (pure) Prolog data structure. Null-ary functor (atom): `Zero`. Unary functor: `Succ`.

```
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

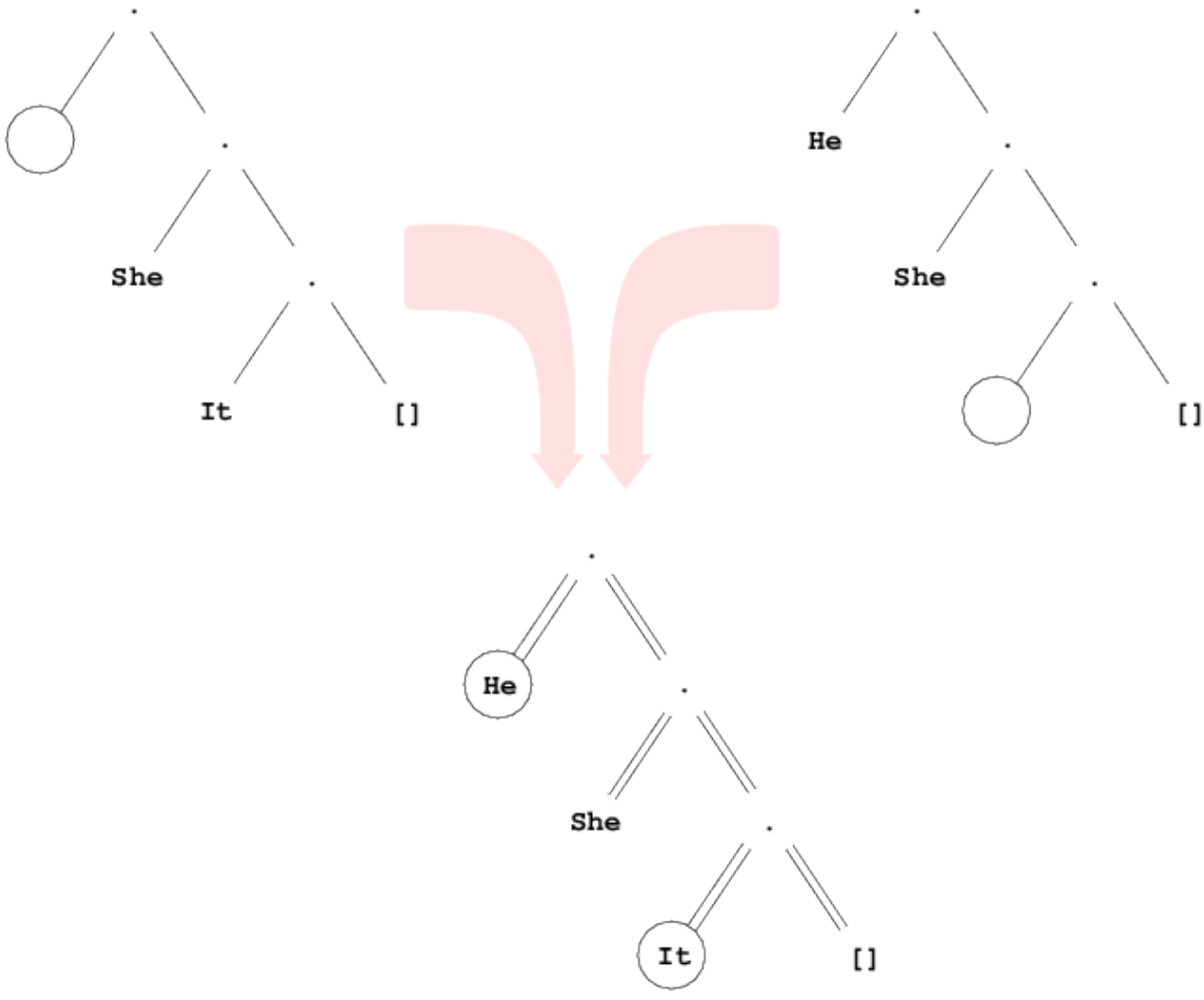
```
Female (Alice)          Female (Alice)
Female (Alice)          Female (Victoria)
Female (Alice)          Male (Albert)

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)
```


.
.
She
.
It
[]
.
He
.
She
.
[]
.
He
.
She
.
It
[]

Unification

term	term	result
`C(x,C(y,C(z,N)))`	`C(He,C(She,C(It,N)))`	`x=He,y=She,z=It`
`C(We,N)`	`C(x,y)`	`x=We,y=N`
`C(They,C(You,N))`	`C(You,C(x,N))`	fails
`C(x,C(She,C(It,N)))`	`C(He,C(She,C(y,N)))`	`x=He,y=It`

term	term	result
`[x,y,z]`	`[He,She,It]`	`x=He,y=She,z=It`
`[We]`	`[x|y]`	`x=We,y=[]`
`[They,You]`	`[You,x]`	fails
`[x,She,It]`	`[He,She,y]`	`x=He,y=It`

Unification

Unification play an important role in many areas of symbolic computation:

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

Unification

Term. A term is constructed from variables and function symbols, e.g., a, $f(b)$, $g(x, a)$. (Here we ignore the distinction between relation symbols and functor symbols.)
Substitution. A *substitution* is an 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 σ 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 σ*, 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.)

Unification. Two terms t, and s, are said to unify if there is a substitution σ such the value of the two terms are equal (symbol by symbol). $t\sigma = s\sigma$.

Unification

Unifying substitution: a substitution that unifies two terms.
Unification problem: given two terms find a unifying substitution, if one exists.
Most general substitution: a substitution with no irrelevant bindings. (We don't bother to give a precise definition.)

Examples

Suppose σ is

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

t	$t\sigma$

Examples

Suppose σ is

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

t	$t\sigma$
x_1	y

Examples

Suppose σ is

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

t	$t\sigma$
x_1	y
x_2	$f(a)$

Examples

Suppose σ is

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

t	$t\sigma$
x_1	y
x_2	$f(a)$
$f(a)$	$f(a)$

Examples

Suppose σ is

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

t	$t\sigma$
x_1	y
x_2	$f(a)$
$f(a)$	$f(a)$
$f(y)$	$f(y)$

Examples

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

x	a	$\{x \mapsto a\}$
a	z	
x	$f(b)$	
x	$f(y)$	
$f(a)$	y	
$g(z, c)$	x	
x	x	
x	y	
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	
x	$f(y)$	
$f(a)$	y	
$g(z, c)$	x	
x	x	
x	y	
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	
$f(a)$	y	
$g(z, c)$	x	
x	x	
x	y	
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	
$g(z, c)$	x	
x	x	
x	y	
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	$\{y \mapsto f(a)\}$
$g(z, c)$	x	
x	x	
x	y	
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	$\{y \mapsto f(a)\}$
$g(z, c)$	x	$\{x \mapsto g(z, c)\}$
x	x	
x	y	
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	$\{y \mapsto f(a)\}$
$g(z, c)$	x	$\{x \mapsto g(z, c)\}$
x	x	$\{\}$ the empty or id subst
x	y	
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	$\{y \mapsto f(a)\}$
$g(z, c)$	x	$\{x \mapsto g(z, c)\}$
x	x	$\{\}$ the empty or id subst
x	y	$\{x \mapsto y\}$ or $\{y \mapsto x\}$
$g(a, b)$	$g(a, b)$	
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	$\{y \mapsto f(a)\}$
$g(z, c)$	x	$\{x \mapsto g(z, c)\}$
x	x	$\{\}$ the empty or id subst
x	y	$\{x \mapsto y\}$ or $\{y \mapsto x\}$
$g(a, b)$	$g(a, b)$	$\{\}$ the empty or id subst
$g(a, x)$	$g(b, x)$	
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	$\{y \mapsto f(a)\}$
$g(z, c)$	x	$\{x \mapsto g(z, c)\}$
x	x	$\{\}$ the empty or id subst
x	y	$\{x \mapsto y\}$ or $\{y \mapsto x\}$
$g(a, b)$	$g(a, b)$	$\{\}$ the empty or id subst
$g(a, x)$	$g(b, x)$	no unifer $a \neq b$
$g(a, x)$	$h(a, b)$	

Examples

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

x	a	$\{x \mapsto a\}$
a	z	$\{z \mapsto a\}$
x	$f(b)$	$\{x \mapsto f(b)\}$
x	$f(y)$	$\{x \mapsto f(y)\}$
$f(a)$	y	$\{y \mapsto f(a)\}$
$g(z, c)$	x	$\{x \mapsto g(z, c)\}$
x	x	$\{\}$ the empty or id subst
x	y	$\{x \mapsto y\}$ or $\{y \mapsto x\}$
$g(a, b)$	$g(a, b)$	$\{\}$ the empty or id subst
$g(a, x)$	$g(b, x)$	no unifer $a \neq b$
$g(a, x)$	$h(a, b)$	no unifer $g \neq h$

Examples

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

$g(x, g(y, z))$	$g(a, g(f(b), c))$	$\{x \mapsto a,\ y \mapsto f(b),\ z \mapsto c\}$
$g(x, a)$	$g(c, y)$	
$f(g(c, x))$	$f(g(y, d))$	
$g(f(x), h(y, b))$	$g(z, x)$	
$j(x, b, y)$	$j(y, b, c)$	
$j(x, b, c)$	$j(a, b, x)$	

Examples

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

$g(x, g(y, z))$	$g(a, g(f(b), c))$	$\{x \mapsto a,\ y \mapsto f(b),\ z \mapsto c\}$
$g(x, a)$	$g(c, y)$	$\{x \mapsto c,\ y \mapsto a\}$
$f(g(c, x))$	$f(g(y, d))$	
$g(f(x), h(y, b))$	$g(z, x)$	
$j(x, b, y)$	$j(y, b, c)$	
$j(x, b, c)$	$j(a, b, x)$	

Examples

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

$g(x, g(y, z))$	$g(a, g(f(b), c))$	$\{x \mapsto a,\ y \mapsto f(b),\ z \mapsto c\}$
$g(x, a)$	$g(c, y)$	$\{x \mapsto c,\ y \mapsto a\}$
$f(g(c, x))$	$f(g(y, d))$	$\{x \mapsto d,\ y \mapsto c\}$
$g(f(x), h(y, b))$	$g(z, x)$	
$j(x, b, y)$	$j(y, b, c)$	
$j(x, b, c)$	$j(a, b, x)$	

Examples

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

$g(x, g(y, z))$	$g(a, g(f(b), c))$	$\{x \mapsto a,\ y \mapsto f(b),\ z \mapsto c\}$
$g(x, a)$	$g(c, y)$	$\{x \mapsto c,\ y \mapsto a\}$
$f(g(c, x))$	$f(g(y, d))$	$\{x \mapsto d,\ y \mapsto c\}$
$g(f(x), h(y, b))$	$g(z, x)$	$\{z \mapsto f(h(y, b)),\ x \mapsto h(y, b)\}$
$j(x, b, y)$	$j(y, b, c)$	
$j(x, b, c)$	$j(a, b, x)$	

Examples

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

$g(x, g(y, z))$	$g(a, g(f(b), c))$	$\{x \mapsto a,\ y \mapsto f(b),\ z \mapsto c\}$
$g(x, a)$	$g(c, y)$	$\{x \mapsto c,\ y \mapsto a\}$
$f(g(c, x))$	$f(g(y, d))$	$\{x \mapsto d,\ y \mapsto c\}$
$g(f(x), h(y, b))$	$g(z, x)$	$\{z \mapsto f(h(y, b)),\ x \mapsto h(y, b)\}$
$j(x, b, y)$	$j(y, b, c)$	$\{x \mapsto c,\ y \mapsto c\}$
$j(x, b, c)$	$j(a, b, x)$	

Examples

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

$g(x, g(y, z))$	$g(a, g(f(b), c))$	$\{x \mapsto a,\ y \mapsto f(b),\ z \mapsto c\}$
$g(x, a)$	$g(c, y)$	$\{x \mapsto c,\ y \mapsto a\}$
$f(g(c, x))$	$f(g(y, d))$	$\{x \mapsto d,\ y \mapsto c\}$
$g(f(x), h(y, b))$	$g(z, x)$	$\{z \mapsto f(h(y, b)),\ x \mapsto h(y, b)\}$
$j(x, b, y)$	$j(y, b, c)$	$\{x \mapsto c,\ y \mapsto c\}$
$j(x, b, c)$	$j(a, b, x)$	no unifer

Unification

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

$\underline{s(x, g(f(z), v, a))}$ $\underline{s(f(y), g(x, h(x), y))}$ s matches

Unification

$\underline{s(x, g(f(z), v, a))}$ $\quad$ $\underline{s(f(y), g(x, h(x), y))}$ $\quad$ s matches

$s(\underline{x}, g(f(z), v, a))$ $\quad$ $s(\underline{f(y)}, g(x, h(x), y))$

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	substitute

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	substitute
$s(f(y), g(f(y), h(f(y)), \underline{a}))$	$s(f(y), g(f(y), h(f(y)), \underline{y}))$	

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	substitute
$s(f(y), g(f(y), h(f(y)), \underline{a}))$	$s(f(y), g(f(y), h(f(y)), \underline{y}))$	$y \mapsto a$

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	substitute
$s(f(y), g(f(y), h(f(y)), \underline{a}))$	$s(f(y), g(f(y), h(f(y)), \underline{y}))$	$y \mapsto a$
$s(f(a), g(f(a), h(f(a)), \underline{a}))$	$s(f(a), g(f(a), h(f(a)), \underline{a}))$	

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	substitute
$s(f(y), g(f(y), h(f(y)), \underline{a}))$	$s(f(y), g(f(y), h(f(y)), \underline{y}))$	$y \mapsto a$
$s(f(a), g(f(a), h(f(a)), \underline{a}))$	$s(f(a), g(f(a), h(f(a)), \underline{a}))$	substitute

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	substitute
$s(f(y), g(f(y), h(f(y)), \underline{a}))$	$s(f(y), g(f(y), h(f(y)), \underline{y}))$	$y \mapsto a$
$s(f(a), g(f(a), h(f(a)), \underline{a}))$	$s(f(a), g(f(a), h(f(a)), \underline{a}))$	substitute

Unification

$\underline{s(x, g(f(z), v, a))}$	$\underline{s(f(y), g(x, h(x), y))}$	s matches
$s(\underline{x}, g(f(z), v, a))$	$s(\underline{f(y)}, g(x, h(x), y))$	$x \mapsto f(y)$
$s(\underline{f(y)}, g(f(z), v, a))$	$s(\underline{f(y)}, g(f(y), h(f(y)), y))$	substitute
$s(f(y), \underline{g(f(z), v, a)})$	$s(f(y), \underline{g(f(y), h(f(y)), y)})$	g matches
$s(f(y), g(\underline{f(z)}, v, a))$	$s(f(y), g(\underline{f(y)}, h(f(y)), y))$	f matches
$s(f(y), g(f(\underline{z}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	$z \mapsto y$
$s(f(y), g(f(\underline{y}), v, a))$	$s(f(y), g(f(\underline{y}), h(f(y)), y))$	substitute
$s(f(y), g(f(y), \underline{v}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	$v \mapsto h(f(y))$
$s(f(y), g(f(y), \underline{h(f(y))}, a))$	$s(f(y), g(f(y), \underline{h(f(y))}, y))$	substitute
$s(f(y), g(f(y), h(f(y)), \underline{a}))$	$s(f(y), g(f(y), h(f(y)), \underline{y}))$	$y \mapsto a$
$s(f(a), g(f(a), h(f(a)), \underline{a}))$	$s(f(a), g(f(a), h(f(a)), \underline{a}))$	substitute

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

t	$t\sigma$
b	b
$f(y)$	$f(y)$
$f(x)$	$f(g(b, x))$
$f(g(b, x))$	$f(g(b, g(b, x)))$

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 Languaages – II*.

Implementations offering sound unification for all unifications are Qu-Prolog and Strawberry Prolog and (optionally, via a runtime flag): XSB and SWI-Prolog.

∗ Idempotence

A unary operation is said to be *idempotent* if, whenever it is applied twice to any element, it gives the same result as if it were applied once.

If f is a function from some set X into itself, then f is idempotent if, for all x in X,

$$f(f(x)) = f(x)$$

Unification Algorithm

Initialize θ 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 θ
      add $X \mapsto Y$ to θ
    Y is a variable not occurring in X:
      substitute X for Y in stack and θ
      add $Y \mapsto X$ to θ
    $X = f(X_1, \ldots, X_n)$ and $Y = f(Y_1, \ldots, Y_n)$:
      push $X_i = Y_i$ on the stack
    otherwise, fail.

Unification Algorithm

- linear time $O(n)$
  M. S. Paterson and M. N. Wegman. "Linear unification." In *Proceedings of the Eighth Annual Symposium of Theory of Computing*, 1976, pages 181–186.
- near linear time $O(nA^{-1}(n))$, union-find
  A. Martelli and U. Montanari. "An efficient unification algorithm." *ACM Transactions on Programming Languages and Systems*, April 1982, pages 258-282.

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.

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

Prolog finds a solutions, 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:

$$(Q_1 \& \cdots \& Q_k) \Rightarrow (P_1 \mid \cdots \mid P_n)$$

(Universal quantification is implicit, existential quantification through skolemization.)

∗ Clausal Form

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

$\bot$	$\bot \Leftarrow \top$
$A \mid B$	$A, B \Leftarrow \top$
$A \mid \neg B$	$A \Leftarrow B$
$\neg A$	$\bot \Leftarrow A$
$\neg(A \& B)$	$\bot \Leftarrow A, B$
$A \& B$	$A \Leftarrow \top$ $B \Leftarrow \top$
$(A \& B) \Rightarrow (C \mid D)$	$C, D \Leftarrow A, B$
$\neg A \mid \neg B \mid C \mid D$	$C, D \Leftarrow A, B$

∗ Resolution

Robinson's resolution principle:

$$\frac{Q_1, \ldots, Q_n \Leftarrow P_1, P_2, \ldots, P_k \qquad R_1, R_2, \ldots, R_m \Leftarrow S_1 \ldots, S_l}{(Q_1, \ldots, Q_n, R_2, \ldots, R_m)\sigma \Leftarrow (P_2, \ldots, P_k, S_1, \ldots, S_n)\sigma}$$

where P_1 and R_1 unify under the most general unifying substitution σ.

One rule of inference for first-order predicate logic.

John Alan Robinson. "A machine-oriented logic based on the resolution principle." *Journal of the ACM*, volume 12, number 1, January 1965, pages 34–41.

∗ 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:

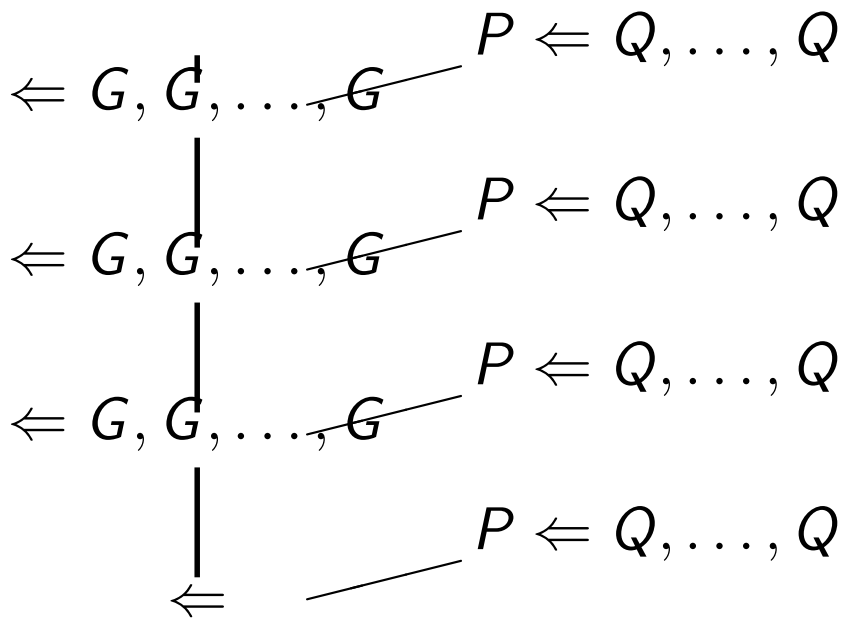
$$\frac{\Leftarrow Q_1, Q_2, \ldots, Q_n \qquad P_0 \Leftarrow P_1, P_2, \ldots, P_k}{\Leftarrow (P_1, \ldots, P_k, Q_2, \ldots, Q_n)\sigma}$$

where Q_1 and P_0 unify under the most general unifying substitution σ.

Resolution principle in other notation:

$$\frac{\neg(Q_1 \& \ldots \& Q_n) \qquad P_1 \& \ldots \& P_k \Rightarrow P_0}{\neg(P_1 \& \ldots \& P_k \& Q_2 \& \ldots \& Q_n)\sigma}$$

$*$ Linear Input Resolution



* Negation in Prolog

Negation as failure

```
not(C) :- call(C),!,fail.
not(C).
```

Not logically correct

```
P(a).
Q(x) :- not (P(b)).
Q(b)?   /* succeeds because P(b) 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 $\neg P(b)$.
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.