Declarative Programming

1: introduction

Coen De Roover - 2010

1

Acknowledgements

These slides are based on:

slides by Prof. Dirk Vermeir for the same course
http://tinf2.vub.ac.be/~dvermeir/courses/logic_programming/lp.pdf

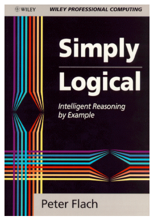
slides by Prof. Peter Flach accompanying his book "Simply Logical"
http://www.cs.bris.ac.uk/~flach/SL/slides/

slides on Computational Logic by the CLIP group
http://clip.dia.fi.upm.es/~logalg/

2

Practicalities

course material



exam

oral test with written preparation about theory and exercises

individual programming project

averaged, unless one ≤ 7

website

http://soft.vub.ac.be/~cderoove/declarative_programming/

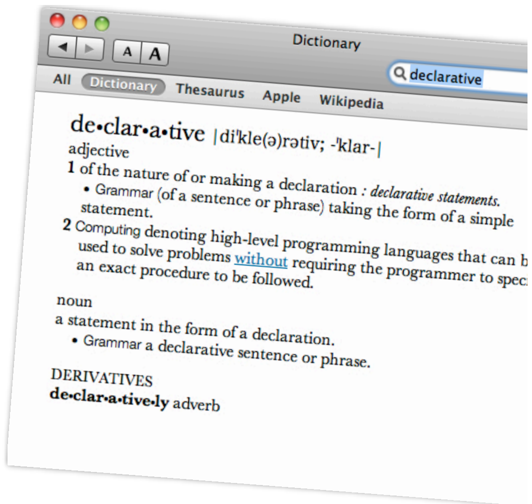
exercises

5 sessions
start 6th of October at IG

3

Problem declaration

Problem solving strategy



Declarative

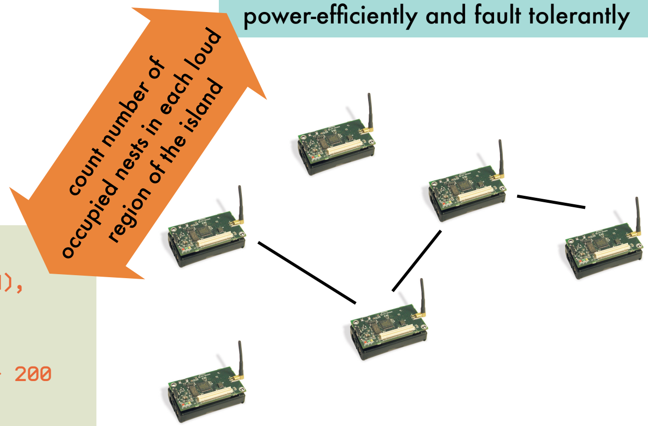
4

Habitat Monitoring using Sensor Network

gather sensor readings

route through network while adjusting averages and count

power-efficiently and fault tolerantly



TinyDB

```
SELECT region,
       CNT(occupied),
       AVG(sound)
FROM sensors
GROUP BY region
HAVING AVG(sound) > 200
EPOCH DURATION 10s
```


program transformations

Jetbrain's SSR

```
if($condition$){
   $x$ = $expr1$;
}
else {
   $x$ = $expr2$;
}
==>
$x$ = $condition$ ? $expr1$ : $expr2$;
```

identifying XML elements

XPath

```
/bookstore/book[price>35.00]/title

/bookstore/book[position()<3]

count(//a[@href]

//img[not(@alt)]
```

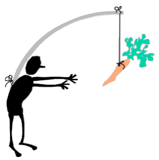
positioning GUI widgets

XWT

```
<Shell>
    <Shell.layout>
       <FillLayout/>
    </Shell.layout>
    <Button text="Hello, world!">
    </Button>
</Shell>
```


also ..

General-purpose declarative programming: logic formalizes human thought process



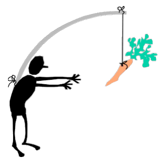
classical logic

```
Aristotle likes cookies
Plato is a friend of anyone who likes cookies
Plato is therefore a friend of Aristotle
```

formally

```
a1 : likes(aristotle, cookies)
a2 : ∀X likes(X, cookies) → friend(plato, X)
t1 :friend(plato,aristotle)
T[a1,a2] ⊢ t1
```


General-purpose declarative programming: logic assertions as problem specification



squares of natural numbers ≤ to 5

extensionally

intensionally

Peano encoding natural numbers	`nat(0) ∧ nat(s(0)) ∧ nat(s(s(0))) ∧ . . .`
	`nat(0) ∧` `∀X : nat(X) → nat(s(X)))`
le	`∀X (le(0,X)) ∧` `∀X,Y (le(X,Y) → le(s(X),s(Y))`
add	`∀X (nat(X) → add(0, X, X)) ∧` `∀X,Y,Z (add(X, Y, Z) → add(s(X), Y, s(Z)))`
prod	`∀X (nat(X) → mult(0, X, 0)) ∧` `∀X,Y,Z,W (mult(X,Y,W) ∧ add(W,Y,Z) → mult(s(X),Y,Z))`
squares	`∀X,Y (nat(X) ∧ nat(Y) ∧ mult(X,X,Y) → square(X,Y))`
wanted	`∀X wanted(X) ←` `(∃Y nat(Y) ∧ le(Y,s(s(s(s(s(0)))))) ∧ square(Y, X)))`

General-purpose declarative programming: proof procedure as problem solver

Assuming the existence of a mechanical proof procedure, a new view of problem solving and computing is possible

[Greene in 60's]

1. program proof procedure once
2. specify the problem by means of logic assertions
3. query the proof procedure for answers that follow from the assertions

query	answer
`nat(s(0)) ?`	`<yes>`
`∃X add(s(0),s(s(0)),X) ?`	`X = s(s(s(0)))`
`∃X wanted(X) ?`	`X=0 ∨ X=s(0) ∨ X=s(s(s(s(0)))) ∨ X=s9(0) ∨ X=s16(0) ∨ X=s25(0)`

General-purpose declarative programming: logic and proof procedure

which logic	which proof procedure
expressivity p versus p(X) logics of quantified truth logics of qualified truth ...	**performance** concurrency, memoization .. **soundness** are all provables true **completeness** can all trues be proven

General-purpose declarative programming: historical overview

60

Greene: problem solving.
Robinson: linear resolution.

70

(early) Kowalski: procedural interpretation of Horn clause logic. Read: A if B_1 and B_2 and … and B_n as: to solve (execute) A, solve (execute) B_1 and B_2 and,..., B_n

(early) Colmerauer: specialized theorem prover (Fortran) embedding the procedural interpretation: Prolog (Programmation et Logique).

In the U.S.: "next-generation AI languages" of the time (i.e. planner) seen as inefficient and difficult to control.

(late) D.H.D. Warren develops DEC-10 Prolog compiler, almost completely written in Prolog. Very efficient (same as LISP). Very useful control builtins.

General-purpose declarative programming: historical overview

80-90

Major research in the basic paradigms and advanced implementation techniques: Japan (Fifth Generation Project), US (MCC), Europe (ECRC, ESPRIT projects).

Numerous commercial Prolog implementations, programming books, and a *de facto* standard, the Edinburgh Prolog family.

First parallel and concurrent logic programming systems.

CLP – Constraint Logic Programming: Major extension – many new applications areas.

1995: ISO Prolog standard.

now

Many commercial CLP systems with fielded applications.

Extensions to full higher order, inclusion of functional programming, ...

Highly optimizing compilers, automatic parallelism, automatic debugging.

Concurrent constraint programming systems.

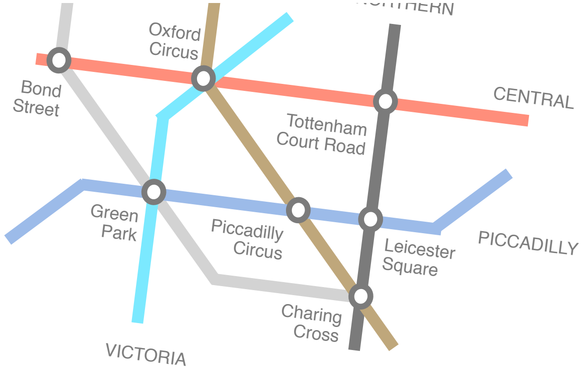
Distributed systems.

Object oriented dialects.

Applications

- Natural language processing
- Scheduling/Optimization problems
- AI related problems
- (Multi) agent systems programming.
- Program analyzers
- ...

Representing Knowledge



relations among underground stations represented by **predicates**

predicate symbol | argument terms

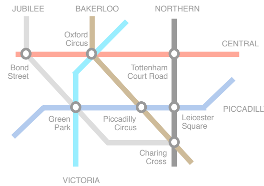
ternary connected/3:

```
connected(bond_street,oxford_circus,central)
...
```

binary nearby/2:

```
nearby(bondstreet,oxford_circus)
...
```


Representing Knowledge: *base information*

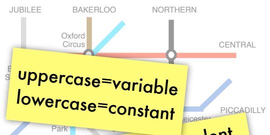


logic predicate connected/3
implemented through logic facts

```
connected(bond_street,oxford_circus,central).
connected(oxford_circus,tottenham_court_road,central).
connected(bond_street,green_park,jubilee).
connected(green_park,charing_cross,jubilee).
connected(green_park,piccadilly_circus,piccadilly).
connected(piccadilly_circus,leicester_square,piccadilly).
connected(green_park,oxford_circus,victoria).
connected(oxford_circus,piccadilly_circus,bakerloo).
connected(piccadilly_circus,charing_cross,bakerloo).
connected(tottenham_court_road,leicester_square,northern).
```

logic facts describe a relation extensionally (i.e., by enumeration)

Representing Knowledge: *derived information*



uppercase=variable
lowercase=constant

variable X in rule 1 is independent from variable X in rule 2

logic predicate nearby/2
implemented through logic rules

"Two stations are *nearby* if they are on the same line with at most one other station in between"

conclusion of rule | premises of rule

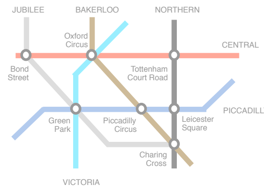
```
nearby(X,Y) :- connected(X,Z,L), connected(Z,Y,L).
nearby(X,Y) :- connected(X,Y,L).
```

logic rules describe a relation intensionally

compare with an extensional description through logic facts:

```
nearby(bond_street,oxford_circus).
nearby(oxford_circus,tottenham_court_road).
nearby(bond_street,tottenham_court_road).
...
```


Answering Queries: *base information*



matching query predicate against a compatible logic fact yields a set of variable bindings

predicate symbol | logic variables as argument terms

query

```
?- connected(W, picadilly_circus, L)
```

answer

```
{ W = green_park, L = picadilly }
```

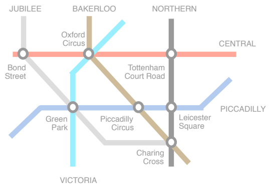
answer

```
{ W = oxford_circus, L = bakerloo }
```

compatible facts

```
...
connected(green_park,piccadilly_circus,piccadilly)
connected(oxford_circus,piccadilly_circus,bakerloo)
...
```

Answering Queries: *derived information*



query ➜ `?- nearby(tottenham_court_road, W).`

matching query predicate with the conclusion of a compatible rule: `nearby(X,Y) :- connected(X,Y,L).`

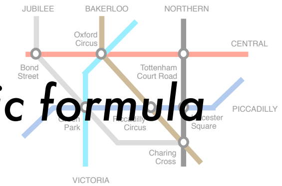
yields: `{ X = tottenham_court_road, Y=W }`

the original query can therefore be answered by answering: `?- connected(tottenham_court_road, W, L).` (premise of compatible rule)

matching new predicate against a compatible logic fact yields: `{ W = leicester_square, L=northern}`

final answer ➜ `{ X = tottenham_court_road, Y = leicester_square }`

Answering a Query = *constructing a proof for a logic formula*



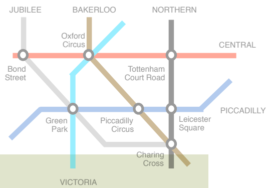

```
?- nearby(tottenham_court_road,W)
   |        nearby(X1,Y1) :- connected(X1,Y1,L1)
   |          { X1=tottenham_court_road, Y1=W }
   v
?- connected(tottenham_court_road,W,L1)
   |     connected(tottenham_court_road, leicester_square)
   |        { W=leicester_square,L1=northern}
   v
   □
```

logic rule (with variables renamed for uniqueness)

logic fact

answer

Answering Queries: *involving recursive rules*




```
reachable(X,Y) :- connected(X,Y,L).
reachable(X,Y) :- connected(X,Z,L), reachable(Z,Y).
```

```
:-reachable(bond_street,W)      reachable(X1,Y1) :- connected(X1,Z1,L1),
   |                                                reachable(Z1,Y1).
   |                              {X1=bond_street, Y1=W}
   v
:-connected(bond_street,Z1,L1),
  reachable(Z1,W)                connected(bond_street,oxford_circus,central).
   |                              {Z1=oxford_circus, L1=central}
   v
:-reachable(oxford_circus,W)     reachable(X2,Y2):-connected(X2,Z2,L2),
   |                                               reachable(Z2,Y2).
   |                              {X2=oxford_circus, Y2=W}
   v
:-connected(oxford_circus,Z2,L2),
  reachable(Z2,W)                connected(oxford_circus,tottenham_court_road,central).
   |                              {Z2=tottenham_court_road, L2=central}
   v
:-reachable(tottenham_court_road,W)
   |                              reachable(X3,Y3) :- connected(X3,Y3,L3).
   |                              {X3=tottenham_court_road, Y3=W}
   v
:-connected(tottenham_court_road,W,L3)
   |                              connected(tottenham_court_road,leicester_square,northern)
   |                              {W=leicester_square, L3=northern}
   v
   □
```

left-most condition expanded first

different rule applications different variables

Prolog's Proof Strategy: *resolution principle*

```
?- nearby(W,charing_cross)
   |      nearby(X1,Y1) :- connected(X1,Z1,L1),
   |                       connected(Z1,Y1,L1).
   |      { X1=W, Y1=charing_cross }
   v
?- connected(W,Z1,L1), connected(Z1,charing_cross,L1)
   |      connected(bond_street, green_park, jubilee).
   |      { W=bond_street, Z1=green_park, L1=jubilee }
   v
?- connected(green_park, charing_cross, jubilee)
   |      connected(green_park, charing_cross, jubilee).
   |      {}
   v
   □
```

resolution principle

to solve a query `?- Q1, ..., Qn`

find a compatible rule `A :- B1, ..., Bm` such that A matches Q_1

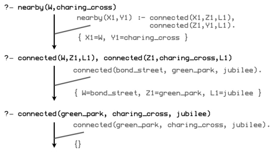
and solve `?- B1, ..., Bm, Q2, ..., Qn`

gives a procedural interpretation to formulas ➠ logic programs

Prolog = programmation en logique

we will investigate where the procedural interpretation of a logic program differs from the declarative one

Prolog's Proof Strategy: *based on proof by refutation*



assume the formula (query) is false and deduce a contradiction

the query `?- nearby(tottenham_court_road,W)`

is answered by reducing `false :- nearby(tottenham_court_road,W)`

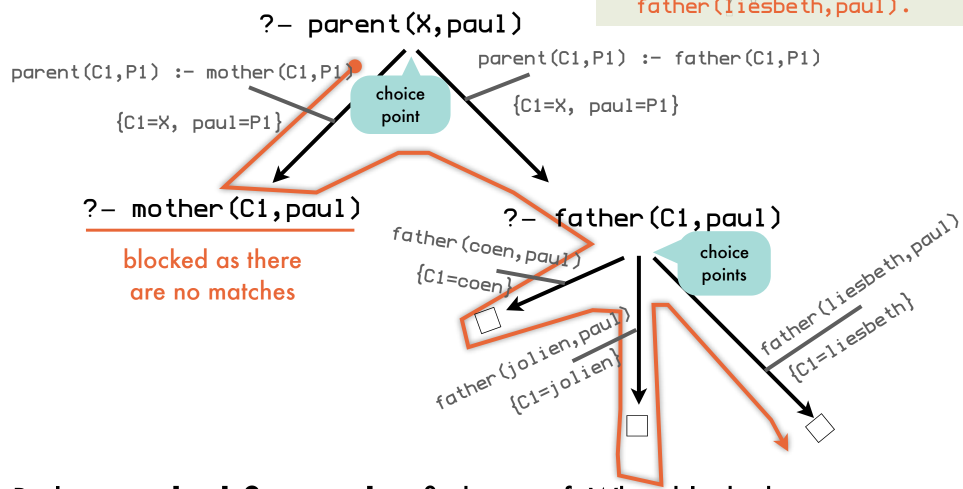
to a contradiction □

"empty rule": premises are always true conclusion is always false

in that case, the query is said "to succeed"

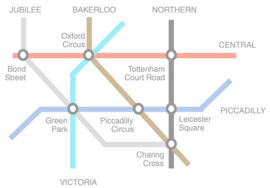
Prolog's Proof Strategy: *searching for a proof*

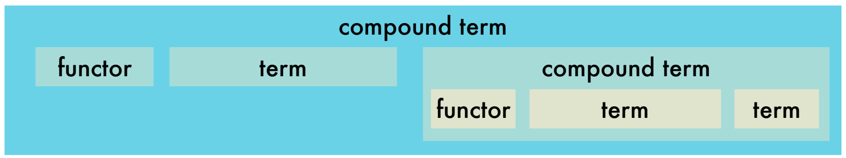
```
parent(C,P):- mother(C,P).
parent(C,P):- father(C,P).

father(coen,paul).
father(jolien,paul).
father(liesbeth,paul).
```



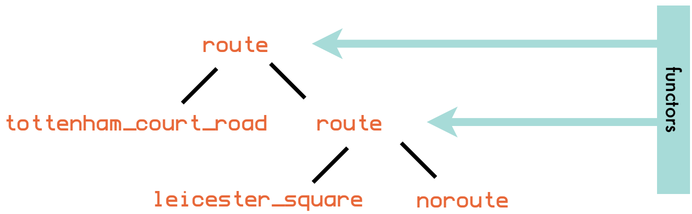
Prolog uses **depth-first search** to find a proof. When blocked or more answers are requested, it **backtracks** to the last choice point. Of multiple conditions, the **left-most** is tried first. Matching rules and facts are tried in the given order.

Representing Knowledge: *compound terms*

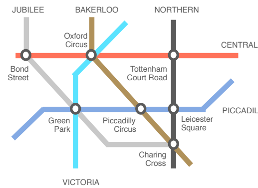




`route(tottenham_court_road, route(leicester_square, noroute))`



Representing Knowledge: *compound terms*




```
reachable(X,Y,noroute):- connected(X,Y,L).

reachable(X,Y,route(Z,R)):- connected(X,Z,L),
                             reachable(Z,Y,R).
```

do not differ syntactically from predicates, but can be used as their arguments

not evaluated in regular logic programming!!

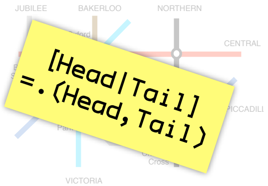
`?- reachable(oxford_circus,charing_cross,R).`

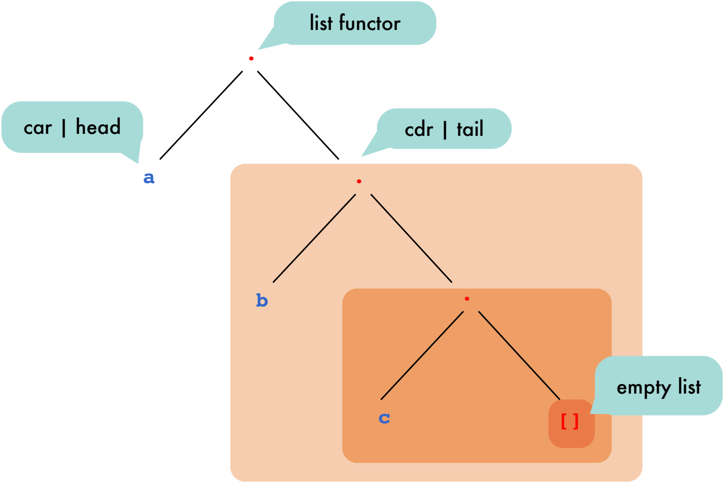
answer `{ R = route(tottenham_court_road, route(leicester_square,noroute)) }`

answer `{ R = route(piccadilly_circus,noroute)}`

answer `{ R = route(piccadilly_circus, route(leicester_square,noroute))}`

Representing Knowledge:
lists



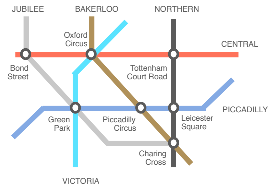


list notations

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

compound term notation → `.(a,.(b,.(c,[])))`

Representing Knowledge:
lists



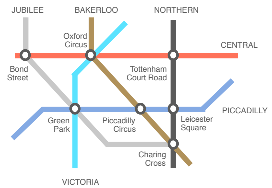
arbitrary length →
```
list([]).
list([First|Rest]) :- list(Rest).
```

even length →
```
evenlist([]).
evenlist([First,Second|Rest]) :- evenlist(Rest).
```

odd length →
```
oddlist([One]).
oddlist([First,Second|Rest]) :- oddlist(Rest).
```

```
oddList([First|Rest]):- evenlist(Rest).
```


Representing Knowledge:
lists




```
reachable(X,Y,[]):- connected(X,Y,L).

reachable(X,Y,[Z|R]):- connected(X,Z,L),
                       reachable(Z,Y,R).
```

```
?- reachable(oxford_circus,charing_cross,R)
```

answer → `{ R= [tottenham_court_road, leicester_square] }`

answer → `{ R =[piccadilly_circus] }`

answer → `{ R =[piccadilly_circus, leicester_square] }`

```
?- reachable(X,charing_cross,[A,B,C,D])
```

from which X can we reach charing_cross via 4 successive intermediate stations A,B,C,D

Illustrative Logic Programs:
list membership

anonymous variable: use when you do not care about the variable's binding

```
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
```

```
?- member(X, [1,2,3])
```

answers → `{ X = 1 }` `{ X = 2 }` `{ X = 3 }`

```
?- member(h(X), [f(1),g(2),h(3)])
```

answer → `{ X = 3 }`

```
?- member(1, [])
```

query fails (the empty list has no members)

Illustrative Logic Programs: *list concatenation*


```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

input → output

```
?- append([a,b,c], [d,e,f], Result)
```

answer → { Result = [a,b,c,d,e,f]}

output → possible inputs

```
?- append(Left, Right, [a,b,c])
```

answer → { Left = [a,b,c,d,e,f], Right= []}

answer → { Left = [a], Right= [b,c]}

answer → { Left = [a,b], Right= [c]}

answer → { Left = [a,b,c], Right= []}

possible because of the relational nature of logic programming

Illustrative Logic Programs: *basic relational algebra*

union

```
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).
```

intersection

```
r_meet_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).
```

cartesian product

```
r_x_s(X1,...,Xm,Xm+1,...,Xm+n) :- r(X1,...,Xm),
                                  s(Xm+1,...,Xm+n).
```

projection

```
r13(X1,X3) :- r(X1,X2,X3).
```

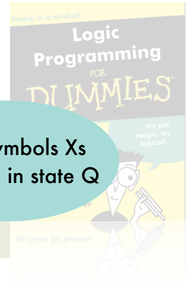
selection

```
r1(X1,X2,X3) :- r(X1,X2,X3), smith_or_jones(X1).
smith_or_jones(smith).
smith_or_jones(jones).
```

natural join

```
r_join_X2_s(X1,X2,...,Xn,Y1,...Yn) :- r(X1,X2...,Xn),
                                      s(X2,Y1,...,Yn)
```


Illustrative Logic Programs: *deterministic finite automaton*

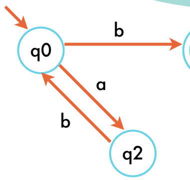



```
accept(Xs) :- initial(Q), accept(Xs,Q).

accept([],Q) :- final(Q).
accept([X|Xs],Q) :- delta(Q,X,Q1), accept(Xs,Q1).
```

list of symbols Xs accepted in state Q

accept/1 ≠ accept/2

transition from state Q to state Q1 consuming X



(ab)*b

```
initial(q0).
final(q1).

delta(q0,b,q1).
delta(q0,a,q2).
delta(q2,b,q0).
```

[The Art of Prolog, Sterling&Shapiro]

accepting

```
?- accept([a, b, a, b, b]).
```

answer → {}

```
?- accept([a, b]).
```

query fails

generating

```
?- accept(Xs).
```

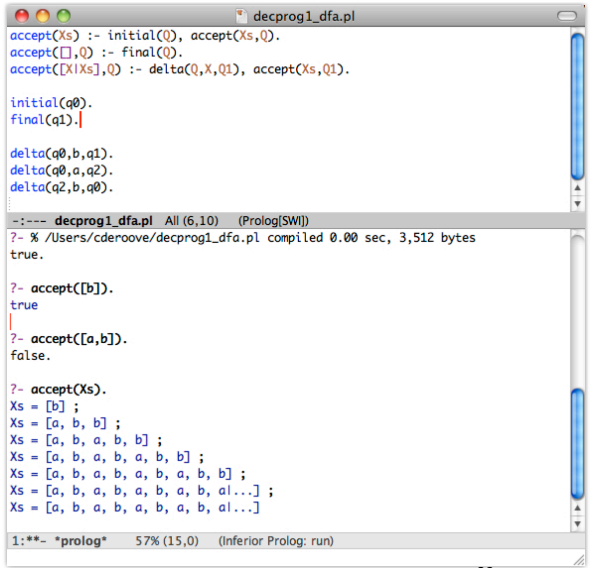
answer → { Xs = [b] }

answer → { Xs = [a,b,b] }

answer → { Xs = [a,b,a,b,b] }

...

Illustrative Logic Programs: *deterministic finite automaton*



```
accept(Xs) :- initial(Q), accept(Xs,Q).
accept([],Q) :- final(Q).
accept([X|Xs],Q) :- delta(Q,X,Q1), accept(Xs,Q1).

initial(q0).
final(q1).

delta(q0,b,q1).
delta(q0,a,q2).
delta(q2,b,q0).
```

-:--- decprog1_dfa.pl All (6,10) (Prolog[SWI])

```
?- % /Users/cderoove/decprog1_dfa.pl compiled 0.00 sec, 3,512 bytes
true.

?- accept([b]).
true

?- accept([a,b]).
false.

?- accept(Xs).
Xs = [b] ;
Xs = [a, b, b] ;
Xs = [a, b, a, b, b] ;
Xs = [a, b, a, b, a, b, b] ;
Xs = [a, b, a, b, a, b, a, b, b] ;
Xs = [a, b, a, b, a, b, a, b, a|...] ;
Xs = [a, b, a, b, a, b, a, b, a|...]
```

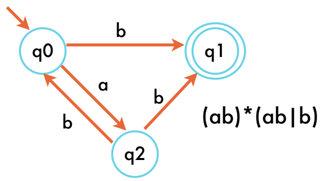
1:**- *prolog* 57% (15,0) (Inferior Prolog: run)

demo time

Illustrative Logic Programs:
non-deterministic finite automaton

for free because of backtracking over choice points

[http://www.cse.buffalo.edu/faculty/alphonce/.OldPages/CPSC312/CPSC312/Lecture/LectureHTML/CS312_10.html#11]




```
initial(q0).
final(q1).

delta(q0,b,q1).
delta(q0,a,q2).
delta(q2,b,q0).
delta(q2,b,q1).
```

accepting

```
?- accept([a,b]).
```

answer → {}

```
?- accept([a,b,b]).
```

query fails

generating

```
?- accept(Xs).
```

answer → { Xs = [b] }

answer → { Xs = [a,b,b] }

answer → { Xs = [a,b,a,b,b] }

...

note that [a,b] is accepted, but not generated ... more about the limitations of the proof procedure later

Illustrative Logic Programs:
non-deterministic pushdown automaton

list used as stack

```
accept(Xs) :- initial(Q), accept(Xs,Q,[]).

accept([],Q,[]) :- final(Q).
accept([X|Xs],Q,S) :- delta(Q,X,S,Q1,S1), accept(Xs,Q1,S1).
```

from state Q with stack S to state Q1 with stack S1 consuming X

[The Art of Prolog, Sterling&Shapiro]

palindrome recognizer

```
initial(q0).
final(q1).
delta(q0,X,S,q0, [X|S]).
delta(q0,X,S,q1, [X|S]).
delta(q0,X,S,q1,S).
delta(q1,X, [X|S],q1,S).
```

input symbols are pushed

transition for palindromes of even length: abba

transition for palindromes of odd length: madam

symbols are popped and compared with input

X pushed on stack

variable X substitutes for a concrete symbol !!

X popped off stack

Declarative Programming

2: theoretical backgrounds

Logic Systems:
structure and meta-theoretical properties

logic system

syntax

defines which "sentences" are legal in the logical language

semantics

gives a meaning to the sentences

usually truth-functional: what is the truth value of a sentence given the truth value of its words

proof theory

specifies how to obtain new sentences (theorems) from assumed ones (axioms) through inference rules

about / about

soundness

anything you can prove is true

weakest form: prove nothing

completeness

anything that is true can be proven

Logic Systems: *roadmap towards Prolog*

clausal logic

propositional clausal logic

statements that can be true or false

```
married;bachelor :- man,adult.
```

relational clausal logic

statements concern relations among objects from a universe of discourse

```
likes(peter,S):-student_of(S,peter).
```

full clausal logic

compound terms aggregate objects

```
loves(X,person_loved_by(X)).
```

definite clause logic

Pure Prolog

no disjunction in head

lacks control constructs, arithmetic of full Prolog

Propositional Clausal Logic - *Syntax*: clauses

:-	if
;	or
,	and

optional

zero or more

```
clause : head [:- body]
  head : [atom [;atom]*]
  body : atom [,atom]*
  atom : single word starting with lower case
```

"someone is married or a bachelor if he is a man and an adult"

```
married;bachelor:-man,adult.
```


Propositional Clausal Logic - *Syntax*: negative and positive literals of a clause

clause

```
H1;...;Hn :- B1,...,Bm
```

$B \Rightarrow H$

$\equiv \neg B \lor H$

is equivalent to

$H1 \lor \ldots \lor Hn \lor \neg B1 \lor \ldots \lor \neg Bm$

positive literals (H1 … Hn) — negative literals (¬B1 … ¬Bm)

hence a clause can also be defined as a disjunction of literals $L_1 \lor L_2 \lor \ldots \lor L_n$ where each L_i is a literal, i.e. $L_i = A_i$ or $L_i = \neg A_i$, with A_i a proposition.

Propositional Clausal Logic - *Syntax*: logic program

finite set of clauses, each terminated by a period

to be read conjunctively

```
woman;man :- human.
    human :- man.
    human :- woman.
```

is equivalent to

$(human \Rightarrow (woman \lor man))$
$\land (man \Rightarrow human)$
$\land (woman \Rightarrow human)$

$(\neg human \lor woman \lor man)$
$\land (\neg man \lor human)$
$\land (\neg woman \lor human)$

$B \Rightarrow H$

$\equiv \neg B \lor H$

Propositional Clausal Logic - *Syntax*: special clauses

an **empty body** stands for **true**	an **empty head** stands for **false**
`man :-.` or `man.`	`:- impossible.`
`true ⇒ man`	`impossible ⇒ false`

`man ∧ ¬impossible`

Propositional Clausal Logic - *Semantics*: Herbrand base, interpretation and models

Herbrand base B_P of a program P

set of all atoms occurring in P

when represented by the set of true propositions I: subset of Herband base

Herbrand interpretation i of P

mapping from Herbrand base B_P to the set of truth values

`i : BP → {true, false}`

An interpretation is a **model for a clause** if the clause is true under the interpretation.

if either the head is true or the body is false

H	B	H:-B
true	true	true
false	true	false
true	false	true
false	false	true

An interpretation is a **model for a program** if it is a model for each clause in the program.

Propositional Clausal Logic - *Semantics*: example (1)

program P

```
woman;man :- human.
human :- man.
human :- woman.
```

Herbrand base B_P

`{woman,man,human}`

2^3 possible Herbrand Interpretations

`I={woman}` `J={woman, man}` `K={woman, man, human}`

`L={man}` `M={man, human}`

`N={human}` `O={woman, human}` `P=∅`

`n={(woman, false), (man, false), (human, false)}`

Propositional Clausal Logic - *Semantics*: example (2)

program P

```
woman;man :- human.
human :- man.
human :- woman.
```

`H1 ∨...∨ Hn ∨ ¬B1 ∨...∨ ¬Bm`

for all clauses: either one atom in head is true or one atom in body is false

4 Herbrand interpretations are models for the program

~~`I={woman}`~~ ~~`J={woman, man}`~~ `K={woman, man, human}`

~~`L={man}`~~ `M={man, human}`

~~`N={human}`~~ `O={woman, human}` `P=∅`

Propositional Clausal Logic - *Semantics*: entailment

P entails C

P ⊨ C

clause C is a **logical consequence** of program P
if every model of P is also a model of C

program P

```
woman.
woman;man :- human.
human :- man.
human :- woman.
```

P ⊨ human

models of P

```
J = {woman,man,human}

I = {woman,human}
```

intuitively preferred: doesn't assume anything to be true that doesn't *have* to be true

Propositional Clausal Logic - *Semantics*: minimal models

no subset is a model itself

could define best model to be the minimal one

BUT

```
woman;man :- human.
human.
```

has 3 models of which 2 are minimal

```
K = {woman, human}
L = {man, human}
M = {woman,man,human}
```

clauses have at most one atom in the head

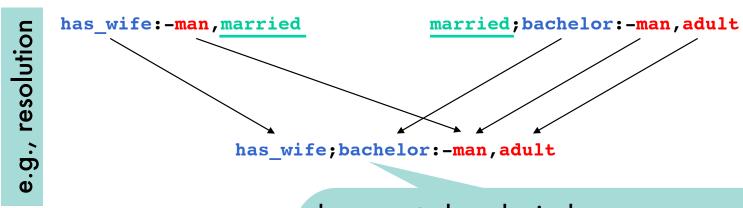
A definite logic program has a unique minimal model.

Propositional Clausal Logic - *Proof Theory*: inference rules

how to check that P ⊨ C without computing all models for P and checking that each is a model for C?

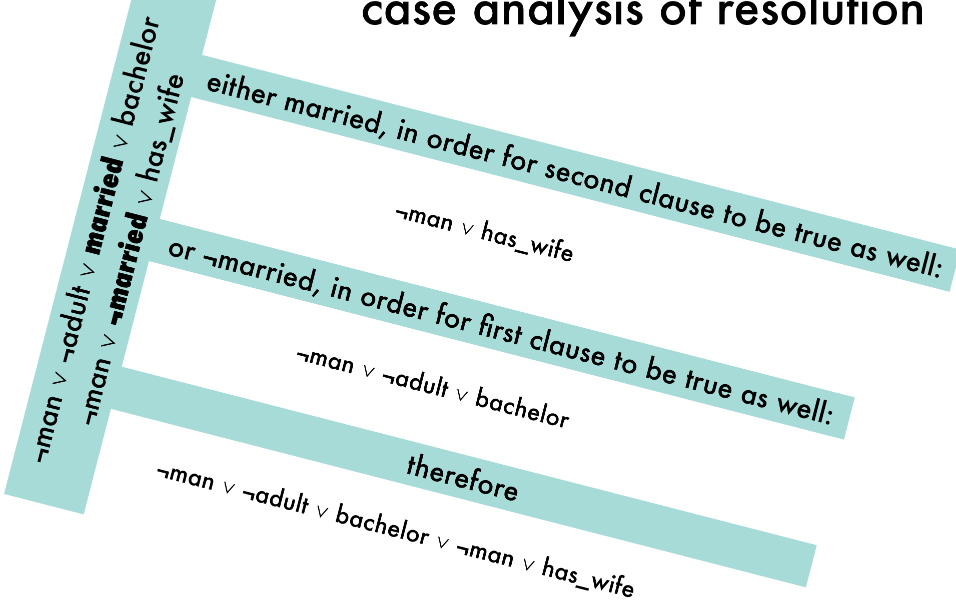
by applying inference rules, C can be derived from P: P ⊢ C

purely syntactic, not concerned with semantics

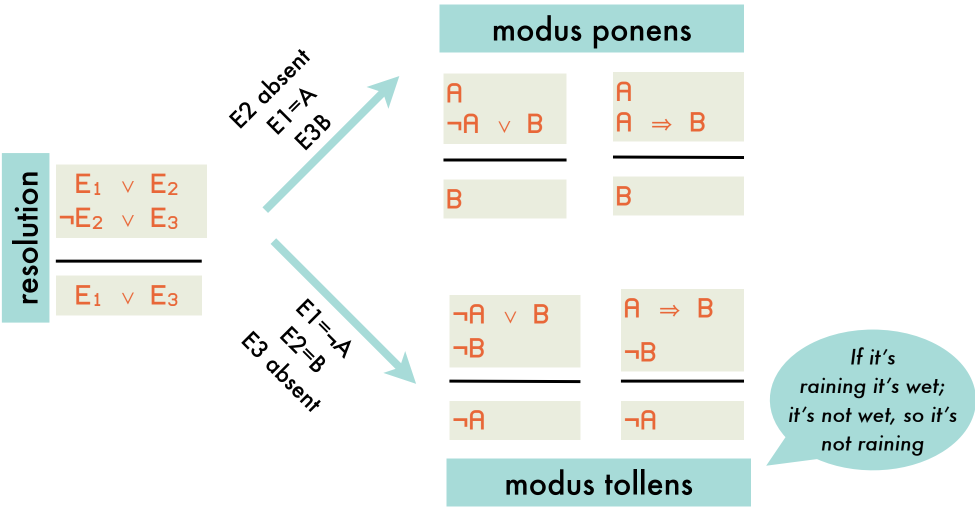


happens to be a logical consequence of the program consisting of both input clauses

Propositional Clausal Logic - *Proof Theory*: case analysis of resolution



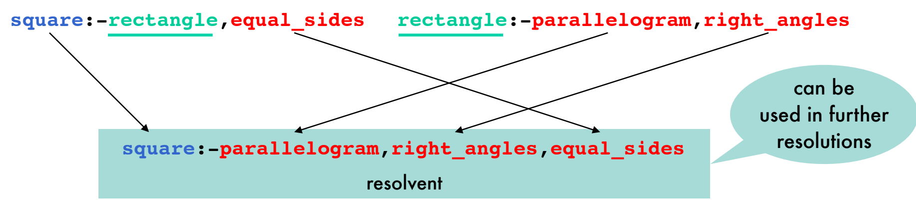
Propositional Clausal Logic - *Proof Theory*: special cases of resolution



Propositional Clausal Logic - *Proof Theory*: successive applications of the resolution inference rule

A proof or derivation of a clause C from a program P
is a sequence of clauses $C_0,\dots,C_n=C$
such that $\forall i_{0\dots n}$: either $C_i \in P$ or C_i is the resolvent of C_{i1} and C_{i2} ($i_1 < i, i_2 < i$).

If there is a proof of C from P, we write $P \vdash C$

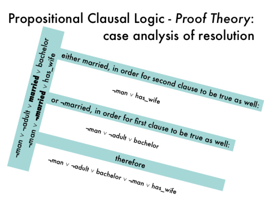


Propositional Clausal Logic - *Meta-theory*: resolution is sound for propositional clausal logic

if $P \vdash C$ then $P \models C$

because every model of the two input clauses is also a model for the resolvent

by case analysis on truth value of resolvent



Propositional Clausal Logic - *Meta-theory*: resolution is incomplete

incomplete

the tautology `a :- a` is true under any interpretation

hence any model for a program P is also a model of `a :- a`

hence $P \models$ `a :- a`

however, resolution cannot establish $P \vdash$ `a :- a`

Propositional Clausal Logic - *Meta-theory*: resolution is refutation-complete

it derives the empty clause from any inconsistent set of clauses

entailment reformulated

P ⊨ C

⇔ each model of P is also a model of C

⇔ no model of P is a model of ¬C

⇔ P∪¬C has no model

P∪¬C is inconsistent

$C = L_1 \vee L_2 \vee \ldots \vee L_n$
$\neg C = \neg L_1 \wedge \neg L_2 \ldots \wedge \neg L_n$
$= \{\neg L_1, \neg L_2 \ldots, \neg L_n\}$
= set of clauses itself

refutation-complete

it can be shown that:

if Q is inconsistent then Q ⊢ □

if P ⊨ C then P∪¬C ⊢ □

empty clause false :- true for which no model exists

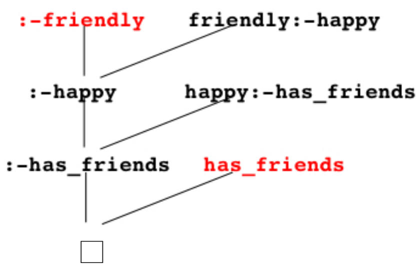
Propositional Clausal Logic - *Meta-theory*: example proof by refutation using resolution

P
```
happy :- has_friends.
friendly :- happy.
```
⊨
```
friendly :- has_friends.
```
C

P∪¬C
```
happy :- has_friends.
friendly :- happy.
has_friends.
:- friendly.
```

=¬(friendly:-has_friends)
=¬(friendly∨¬has_friends)
=¬friendly∧has_friends

P∪¬C ⊢ □



Relational Clausal Logic - *Syntax*: clauses

statements concern relations among objects from a universe of discourse

add constants, variables and predicates to propositional logic

```
 constant : single word starting with lower case
 variable : single word starting with upper case
     term : constant | variable
predicate : single word starting with lower case
     atom : predicate[(term[,term]*)]
   clause : head [:- body]
     head : [atom[;atom]*]
     body : atom[,atom]*
```

"peter likes anybody who is his student. maria is a student of peter"

```
likes(peter,S) :- student_of(S,peter).
student_of(maria,peter).
```


Relational Clausal Logic - *Semantics*: Herbrand universe, base, interpretation

Herbrand universe of a program P

```
{ peter, maria }
```

term without variables

set of all terms that are ground in P

Herbrand base B_P of a program P

```
{ likes(peter,peter),likes(peter,maria),
  likes(maria,peter),likes(maria,maria),
  student_of(peter,peter), student_of(peter,maria),
  student_of(maria,peter), student_of(maria,maria) }
```

set of all ground atoms that can be constructed using predicates in P and arguments in the Herbrand universe of P

Herbrand interpretation I of P

```
{ likes(peter,maria), student_of(maria,peter) }
```

subset of B_P consisting of ground atoms that are true

is this a model? need to consider variable substitutions

Relational Clausal Logic - *Semantics*: substitutions and ground clause instances

A substitution is a mapping σ : Var → Trm.
For a clause C, the result of σ on C, denoted $C\sigma$
is obtained by replacing all occurrences of $X \in$ Var in C by $\sigma(X)$.
$C\sigma$ is an instance of C.

```
if σ={S/maria} then
(likes(peter,S):-student_of(S,peter))σ
=likes(peter,maria):-student_of(maria,peter)
```

Relational Clausal Logic - *Semantics*: models

ground instances of relational clauses are like propositional clauses

interpretation I is a model of a clause C
⇔ I is a model of every ground instance of C.

interpretation I is a model of a program P
⇔ I is a model of each clause $C \in P$.

P
```
likes(peter,S) :- student_of(S,peter).
student_of(maria,peter).
```

I
```
{ likes(peter,maria), student_of(maria,peter) }
```

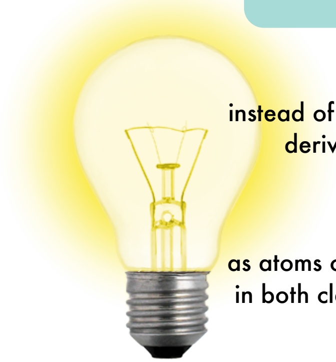
I is a model for P
because it is a model of all ground instances of clauses in P:

```
likes(peter,peter) :- student_of(peter,peter).
likes(peter,maria) :- student_of(maria,peter).
student_of(maria,peter).
```

Relational Clausal Logic - *Proof Theory*: naive version

naive because there are many grounding substitutions, most of which do not lead to a proof

derive the empty clause through propositional resolution from all ground instances of all clauses in P



instead of trying arbitrary substitutions before trying to apply resolution, derive the required substitutions from the literal resolved upon (positive in one clause and negative in the other)

as atoms can contain variables, do not require exactly the same atom in both clauses ... rather a complementary pair of atoms that can be made equal by substituting terms for variables

Relational Clausal Logic - *Proof Theory*: unifier

A substitution σ is a **unifier** of two atoms a_1 and a_2
⇔ $a_1\sigma = a_2\sigma$. If such a σ exists, a_1 and a_2 are called unifiable.

A substitution σ_1 is **more general** than σ_2 if $\sigma_2 = \sigma_1\theta$ for some substitution θ.

A unifier θ of a_1 and a_2 is a **most general unifier** of a_1 and a_2
⇔ it is more general than any other unifier of a_1 and a_2.

If two atoms are unifiable then they their mgu is **unique** up to renaming.

Relational Clausal Logic - *Proof Theory*: unifier examples

```
p(X, b) and p(a, Y) are unifiable
with most general unifier {X/a,Y/b}
```

```
q(a) and q(b) are not unifiable
```

```
q(X) and q(Y) are unifiable:
```

```
{X/Y} (or{Y/X}) is the most general unifier
```

```
{X/a, Y/a} is a less general unifier
```

[http://users.informatik.uni-halle.de/~brass/lp03/c3_purep.pdf]

Relational Clausal Logic - *Proof Theory*: resolution using most general unifier

apply resolution on many clause-instances at once

$$
\begin{aligned}
\text{if}\quad C_1 &= L_1^1 \vee \ldots L_{n_1}^1 \\
C_2 &= L_1^2 \vee \ldots L_{n_2}^2 \\
L_i^1\theta &= \neg L_j^2\theta \quad \text{for some } 1 \leq i \leq n_1,\ 1 \leq j \leq n_2 \\
&\text{where } \theta = \mathbf{mgu}(L_i^1, L_j^2)
\end{aligned}
$$

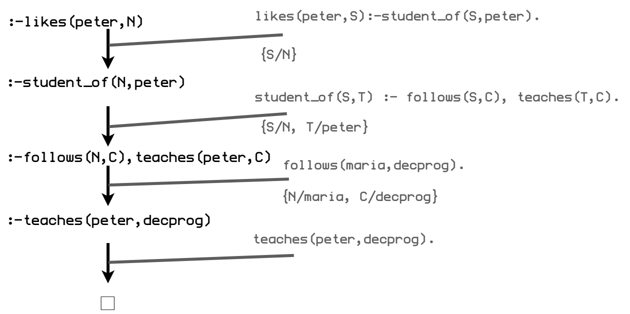
then $L_1^1\theta \vee \ldots \vee L_{i-1}^1\theta \vee L_{i+1}^1\theta \vee \ldots \vee L_{n_1}^1\theta$

$\vee L_1^2\theta \vee \ldots \vee L_{j-1}^2\theta \vee L_{j+1}^2\theta \vee \ldots \vee L_{n_2}^2\theta$

Relational Clausal Logic - *Proof Theory*:
example of proof by refutation using resolution with mgu

P
```
likes(peter,S) :- student_of(S,peter).
student_of(S,T) :- follows(S,C), teaches(T,C).
teaches(peter,decprog).
follows(maria,decprog).
```

"is there anyone whom peter likes"? ➠ add "peter likes nobody" to P



`:- likes(peter,N)){N/maria} ∪ P ⊢ □` hence `P ⊨ likes(peter,maria)`

Relational Clausal Logic - *Meta-theory*: soundness and completeness

sound

relational clausal logic is sound

P⊢C ⇒ P⊨C

complete

relational clausal logic is refutation-complete

P∪{C} inconsistent ⇒ P ∪ {C} ⊢ □

new formulation because
:- p(X).≡∀X·¬p(X)
while ¬(p(X).)≡¬(∀X·p(X))≡∃X·¬p(X)

Relational Clausal Logic - *Meta-theory*: decidability

The question "P⊨C?" is decidable for relational clausal logic.

also for propositional clausal logic

Herbrand universe and base are finite

therefore also interpretations and models

could in principle enumerate all models of P and check whether they are also a model of C

Full Clausal Logic - *Syntax*: clauses

compound terms aggregate objects

Add function symbols (functors), with an arity; constants are 0-ary functors.

```
  functor : single word starting with lower case
 variable : single word starting with upper case
     term : variable | functor[(term[,term]*)]
predicate : single word starting with lower case
     atom : predicate[(term[,term]*)]
   clause : head [:- body]
     head : [atom[;atom]*]
     body : proposition[,proposition]*
```

object

proposition

"adding two Peano-encoded naturals"

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```


Full Clausal Logic - *Semantics*: Herbrand universe, base, interpretation

analogous to relational clausal logic

Herbrand universe of a program P

```
{ 0, s(0), s(s(0)), s(s(s(0))),... }
```

infinite!

terms that can be constructed from the constants and functors

Herbrand base B_P of a program P

```
{ plus(0,0,0), plus(s(0),0,0),
  plus(0,s(0),0), plus(s(0),s(0),0),...}
```

set of all ground atoms that can be constructed using predicates in P and ground terms in the Herbrand universe of P

Herbrand interpretation I of P

is this a model?

```
{ plus(0,0,0), plus(s(0),0,s(0)),plus(0,s(0),s(0))} }
```

possibly infinite subset of B_P consisting of ground atoms that are true

Full Clausal Logic - *Semantics*: infinite models are possible

Herbrand universe is infinite, therefore infinite number of grounding substitutions

An interpretation is a **model for a program** if it is a model for each ground instance of every clause in the program.

```
plus(0,0,0)
plus(s(0),0,s(0)):-plus(0,0,0)
plus(s(s(0)),0,s(s(0))):-plus(s(0),0,s(0))
...
plus(0,s(0),s(0))
plus(s(0),s(0),s(s(0))):-plus(0,s(0),s(s(0)))
plus(s(s(0)),s(0),s(s(s(0)))):-plus(s(0),s(0),s(s(0)))
...
```

according to first ground clause, `plus(0,0,0)` has to be in any model but then the second clause requires the same of `plus(s(0),0,s(0))` and the third clause of `plus(s(s(0)),0,s(s(0)))` ...

all models of this program are necessarily infinite

Full Clausal Logic - *Proof Theory*: computing the most general unifier

analogous to relational clausal logic, but have to take compound terms into acount when computing the mgu of complementary atoms

atoms

`plus(s(0),X,s(X))` and `plus(s(Y),s(0),s(s(Y)))`

have most general unifier

`{Y/0, X/s(0)}`

yields unified atom `plus(s(Y),s(0),s(s(Y)))`

found by

renaming variables so that the two atoms have none in common
ensuring that the atoms' predicates and arity correspond
scanning the subterms from left to right to
- find first pair of subterms where the two atoms differ; (`s(Y)` and `s(0)`)
  - if neither subterm is a variable, unification fails;
  - else substitute the other term for all occurrences of the variable and remember the partial substitution; (`{Y/0}`)
- repeat until no more differences found

Full Clausal Logic - *Proof Theory*: computing the most general unifier using the Martelli-Montanari algorithm

repeat
 select $s = t \in \mathcal{E}$
 case $s = t$ **of**
  $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ $(n \geq 0)$:
   replace $s = t$ **by** $\{s_1 = t_1, \ldots, s_n = t_n\}$
  $f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n)$ $(f/m \neq g/n)$:
   fail
  $X = X$:
   remove $X = X$ **from** $\mathcal{E}$
  $t = X$ $(t \notin$ **Var**$)$:
   replace $t = X$ **by** $X = t$
  $X = t$ $(X \in$ **Var** $\wedge X \neq t \wedge X$ occurs more than once in $\mathcal{E})$:
   if X occurs in t
   then fail
   else replace all occurrences of X in $\mathcal{E}$ (except in $X = t$) by t
 esac
until no change

operates on a finite set of equations s=t

occur check

$\{f(X, g(Y)) = f(g(Z), Z)\}$
$\Rightarrow \{X = g(Z), g(Y) = Z\}$
$\Rightarrow \{X = g(Z), Z = g(Y)\}$
$\Rightarrow \{X = g(g(Y)), Z = g(Y)\}$
$\Rightarrow \{X/g(g(Y)), Z/g(Y)\}$

resulting set = mgu

$\{f(X, g(X), b) = f(a, g(Z), Z)\}$
$\Rightarrow \{X = a, g(X) = g(Z), b = Z\}$
$\Rightarrow \{X = a, X = Z, b = Z\}$
$\Rightarrow \{X = a, a = Z, b = Z\}$
$\Rightarrow \{X = a, Z = a, b = Z\}$
$\Rightarrow \{X = a, Z = a, b = a\}$
$\Rightarrow$ **fail**

Full Clausal Logic - *Proof Theory*: importance of occur check

before substituting a term for a variable, verify that the variable does not occur in the term; if so: fail

program: `loves(X,person_loved_by(X)).`

query: `:- loves(Y,Y).`

without occur check, atoms to be resolved upon unify under substitution

`{Y/X, X/person_loved_by(X)}`

and therefore resolving to the empty clause

no semantics for infinite terms as there are no such terms in the Herbrand base

BUT

try to print answer:

`X=person_loved_by(person_loved_by(person_loved_by(...)))`

moreover, not a logical consequence of the program

omitting occur check renders resolution unsound

Full Clausal Logic - *Proof Theory*: occur check

not performed in Prolog out of performance considerations (e.g. unify X with a list of 1000 elements)

Martelli-Montanari algorithm

$\{l(Y, Y) = l(X, f(X))\}$
$\Rightarrow \{Y = X, Y = f(X)\}$
$\Rightarrow \{Y = X, X = f(X)\}$
$\Rightarrow$ **fail**

SWI-Prolog

```
?- l(Y,Y) = l(X,f(X)).
Y = f(**),
X = f(**).
?-
```

built-in unification operator

```
?- unify_with_occurs_check(l(Y,Y),l(X,f(X))).
false.
?-
```

in rare cases where the occurs check is needed

Full Clausal Logic - *Meta-theory*: soundness, completeness, decidability

sound

full clausal logic is sound

$P \vdash C \Rightarrow P \models C$

complete

full clausal logic is refutation-complete

$P \cup \{C\}$ inconsistent $\Rightarrow P \cup \{C\} \vdash \square$

decidability

The question “$P \models C$?” is only semi-decidable.

> there is no algorithm that will always answer the question (with “yes” or “no”) in finite time; but there is an algorithm that, if $P \models C$, will answer “yes” in finite time but this algorithm may loop if $P \not\models C$.

Clausal Logic: overview

	propositional	relational	full
Herbrand universe	-	`{a,b}` finite	`{a,f(a),f(f(a)),...}` infinite
Herbrand base	`{p, q}`	`{p(a,a), p(b,a),...}`	`{p(a,f(a)), p(f(a), p(f(f(a)),...}`
clause	`p:-q`	`p(X,Z):- q(X,Y),p(Y,Z)`	`p(X,f(X)):- q(X)`
Herbrand models	`{}` `{p}` `{p,q}`	`{}` `{p(a,a)}` `{p(a,a),p(b,a),q(b,a)}` `...` finite number of finite models	`{}` `{p(a,f(a)),q(a)}` `{p(f(a),f(f(a)), q(f(a))} ...` infinite number of finite or infinite models
meta-theory	sound refutation-complete decidable	sound refutation-complete decidable	sound (occurs check) refutation-complete semi-decidable

Clausal Logic: conversion to first-order predicate logic (1)

> Every set of clauses can be rewritten as an equivalent sentence in first-order predicate logic.

> variables in a sentence cannot range over predicates

```
married;bachelor :- man,adult.
haswife :- married.
```

becomes `(man∧adult ⇒ married∨bachelor) ∧ (married ⇒ haswife)`

or `(¬man ∨ ¬adult ∨ married ∨ bachelor ) ∧ (¬married ∨ haswife)`

> $A \Rightarrow B \equiv \neg A \vee B$
> $\neg(A \wedge B) \equiv \neg A \vee \neg B$

> conjunctive normal form: conjunction of disjunction of literals

```
reachable(X,Y,route(Z,R)):- connected(X,Z,L), reachable(Z,Y,R).
```

becomes
```
∀X∀Y∀Z∀R∀L : ¬connected(X,Z,L) ∨
              ¬reachable(Z,Y,R) ∨
              reachable(X,Y,route(Z,R))
```

> variables in clauses are universally quantified

Clausal Logic: conversion to first-order predicate logic (2)

> Every set of clauses can be rewritten as an equivalent sentence in first-order predicate logic.

```
nonempty(X) :- contains(X,Y).
```

becomes `∀X∀Y: nonempty(X)∨¬contains(X,Y)`

or `∀X: (nonempty(X)∨∀Y¬contains(X,Y))`

or `∀X: nonempty(X)∨¬(∃Y:contains(X,Y))`

or `∀X: (∃Y:contains(X,Y))⇒ nonempty(X))`

> variables that occur only in the body of a clause are existentially qualified

Clausal Logic: conversion from first-order predicate logic (1)

For each first order sentence, there exists an “almost equivalent” set of clauses.

```
∀X [brick(X)⇒(∃Y [on(X,Y)∧¬pyramid(Y)] ∧
             ¬∃Y [on(X,Y)  ∧ on(Y,X)] ∧
             ∀Y [¬brick(Y)⇒¬equal(X,Y)]])
```

1. eliminate ⇒ using A ⇒ B ≡ ¬A ∨ B.

```
∀X [¬brick(X)∨(∃Y [on(X,Y)∧¬pyramid(Y)] ∧
              ¬∃Y [on(X,Y)∧on(Y,X)] ∧
               ∀Y [¬(¬brick(Y))∨¬equal(X,Y)]])
```

2. put into negation normal form: negation only occurs immediately before propositions

```
∀X [¬brick(X)∨(∃Y [on(X,Y)∧¬pyramid(Y)] ∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)] ∧
               ∀Y [brick(Y)∨¬equal(X,Y)]])
```

¬(A∧B) ≡ ¬A∨¬B
¬(A∨B) ≡ ¬A∧¬B
¬(¬A) ≡ A
¬∀X [p(X)] ≡ ∃X [¬p(X)]
¬(∃X [p(X)] ≡ ∀X [¬p(X)]

Clausal Logic: conversion from first-order predicate logic (2)

For each first order sentence, there exists an “almost equivalent” set of clauses.

```
∀X [¬brick(X)∨(∃Y [on(X,Y)∧¬pyramid(Y)] ∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)] ∧
               ∀Y [brick(Y)∨¬equal(X,Y)]])
```

model {loves(paul,anna)} can be converted to equivalent {loves(paul,person_loved_by(paul))}

∃X∀Y : loves(X,Y)
Skolem constants substitute for an existentially quantified variable which does not occur in the scope of a universal quantifier

∀X∃Y : loves(X,Y)
∀X:loves(X,person_loved_by(X))

replace existentially quantified variable by a compound term of which the arguments are the universally quantified variables in whose scope the existentially quantified variable occurs

3. replace ∃ using Skolem functors (abstract names for objects, functor has to be new)

```
∀X [¬brick(X)∨( [on(X,sup(X))∧¬pyramid(sup(X))] ∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)] ∧
               ∀Y [brick(Y)∨¬equal(X,Y)]])
```


Clausal Logic: conversion from first-order predicate logic (3)

For each first order sentence, there exists an “almost equivalent” set of clauses.

```
∀X [¬brick(X)∨( [on(X,sup(X))∧¬pyramid(sup(X))] ∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)] ∧
               ∀Y [brick(Y)∨¬equal(X,Y)]])
```

4. standardize all variables apart such that each quantifier has its own unique variable

```
∀X [¬brick(X)∨( [on(X,sup(X))∧¬pyramid(sup(X))] ∧
               ∀Y [¬on(X,Y)∨¬on(Y,X)] ∧
               ∀Z [brick(Z)∨¬equal(X,Z)]])
```

5. move ∀ to the front

```
∀X∀Y∀Z [¬brick(X)∨( [on(X,sup(X))∧¬pyramid(sup(X))] ∧
                   [¬on(X,Y)∨¬on(Y,X)] ∧
                   [brick(Z)∨¬equal(X,Z)]])
```


Clausal Logic: conversion from first-order predicate logic (4)

For each first order sentence, there exists an “almost equivalent” set of clauses.

```
∀X∀Y∀Z [¬brick(X)∨( [on(X,sup(X))∧¬pyramid(sup(X))] ∧
                   [¬on(X,Y)∨¬on(Y,X)] ∧
                   [brick(Z)∨¬equal(X,Z)]])
```

6. convert to conjunctive normal form using A∨(B∧C) ≡ (A∨B)∧(A∨C)

```
∀X∀Y∀Z [(¬brick(X)∨ [on(X,sup(X))∧¬pyramid(sup(X))])∧
        (¬brick(X)∨ [¬on(X,Y)∨¬on(Y,X)])∧
        (¬brick(X)∨ [brick(Z)∨¬equal(X,Z)])]
```

```
∀X∀Y∀Z [((¬brick(X)∨on(X,sup(X)))∧(¬brick(X)∨¬pyramid(sup(X))))∧
        (¬brick(X)∨ [¬on(X,Y)∨¬on(Y,X)])∧
        (¬brick(X)∨ [brick(Z)∨¬equal(X,Z)])]
```

```
∀X∀Y∀Z [ [¬brick(X)∨on(X,sup(X))]∧
        [¬brick(X)∨¬pyramid(sup(X))]∧
        [¬brick(X)∨¬on(X,Y)∨¬on(Y,X)]∧
        [¬brick(X)∨brick(Z)∨¬equal(X,Z)]]
```

A∨(B∨C) ≡ A∨B∨C

Clausal Logic: conversion from first-order predicate logic (5)

For each first order sentence, there exists an "almost equivalent" set of clauses.

```
∀X∀Y∀Z[[¬brick(X)∨on(X,sup(X))]∧
       [¬brick(X)∨¬pyramid(sup(X))]∧
       [¬brick(X)∨¬on(X,Y)∨¬on(Y,X)]∧
       [¬brick(X)∨brick(Z)∨¬equal(X,Z)]]
```

7 split the conjuncts in clauses (a disjunction of literals)

```
∀X   ¬brick(X)∨on(X,sup(X))
∀X   ¬brick(X)∨¬pyramid(sup(X))
∀X∀Y ¬brick(X)∨¬on(X,Y)∨¬on(Y,X)
∀X∀Z ¬brick(X)∨brick(Z)∨¬equal(X,Z)
```

8 convert to clausal syntax (negative literals to body, positive ones to head)

```
on(X,sup(X)) :- brick(X).
:- brick(X), pyramid(sup(X)).
:- brick(X), on(X,Y), on(Y,X).
brick(X) :- brick(Z), equal(X,Z).
```


Clausal Logic: conversion from first-order predicate logic (6)

For each first order sentence, there exists an "almost equivalent" set of clauses.

```
∀X: (∃Y:contains(X,Y))⇒ nonempty(X))
```

1 eliminate ⇒ — `∀X: ¬(∃Y:contains(X,Y))∨nonempty(X))`

2 put into negation normal form — `∀X: (∀Y:¬contains(X,Y))∨nonempty(X))`

3 replace ∃ using Skolem functors

4 standardize variables

5 move ∀ to the front — `∀X∀Y: ¬contains(X,Y)∨nonempty(X)`

6 convert to conjunctive normal form

7 split the conjuncts in clauses

8 convert to clausal syntax — `nonempty(X) :- contains(X,Y)`

Definite Clause Logic: motivation

how to use the clause depends on what you want to prove, but this indeterminacy is a source of inefficiency in refutation proofs

indefinite program

```
married(X);bachelor(X) :- man(X), adult(X).
man(peter). adult(peter). man(paul).
:-married(maria). :-bachelor(maria). :-bachelor(paul).
```

logical consequences that can be derived in two resolution steps

clause is used from right to left

```
married(X);bachelor(X):-man(X),adult(X)        man(peter)
married(peter);bachelor(peter):-adult(peter)   adult(peter)
married(peter);bachelor(peter)
```

indefinite conclusion

clause is used from left to right

```
married(X);bachelor(X):-man(X),adult(X)        :-married(maria)
bachelor(maria):-man(maria),adult(maria)       :-bachelor(maria)
:-man(maria),adult(maria)
```

both literals from head and body are resolved away

```
married(X);bachelor(X):-man(X),adult(X)        man(paul)
married(paul);bachelor(paul):-adult(paul)      :-bachelor(paul)
married(paul):-adult(paul)
```


Definite Clause Logic: syntax and proof procedure

for efficiency's sake

rules out indefinite conclusions → full clausal logic clauses are restricted: at most one atom in the head

A :- $B_1, \ldots, B_n$

fixes direction to use clauses → from right to left: ➠ procedural interpretation

"prove A by proving each of B_i"

Definite Clause Logic: recovering lost expressivity

semantics and proof theory for the not in a general clause will be discussed later; Prolog actually provides a special predicate not/1 which can only be understood procedurally

problem

can no longer express

```
married(X); bachelor(X) :- man(X), adult(X).
man(john). adult(john).
```

characteristic of indefinite clauses

which had two minimal models

```
{man(john),adult(john),married(john)}
{man(john),adult(john),bachelor(john)}
{man(john),adult(john),married(john),bachelor(john)}
```

general clauses

definite clause containing not

first model is minimal model of **general** clause

```
married(X) :- man(X), adult(X), not bachelor(X).
```

second model is minimal model of **general** clause

```
bachelor(X) :- man(X), adult(X), not married(X).
```

to prove that someone is a bachelor, prove that he is a man and an adult, and prove that he is not a bachelor

Declarative Programming

3: logic programming and Prolog

Sentences in definite clause logic: *procedural and declarative meaning*

```
a :- b, c.
```

declarative meaning realized by model semantics

to determine whether a is a logical consequence of the clause, order of atoms in body is irrelevant

procedural meaning realized by proof theory

order of atoms may determine whether a can be derived

`a :- b, c.` to prove a, prove b and then prove c

`a :- c, b.` to prove a, prove c and then prove b

imagine c is false

and proof for b is infinite

Sentences in definite clause logic: *procedural meaning enables programming*

SLD-resolution refutation

procedural knowledge: **how** the inference rules are applied to solve the problem

algorithm = logic + control

declarative knowledge: the **what** of the problem

definite clause logic

SLD-resolution refutation:
turns resolution refutation into a proof procedure

also: an unwieldy theorem prover in effective programming language

left-most

determines how to select a literal to resolve upon

and which clause is used when multiple are applicable

top-down

selection rule

definite clauses

SLD

linear resolution

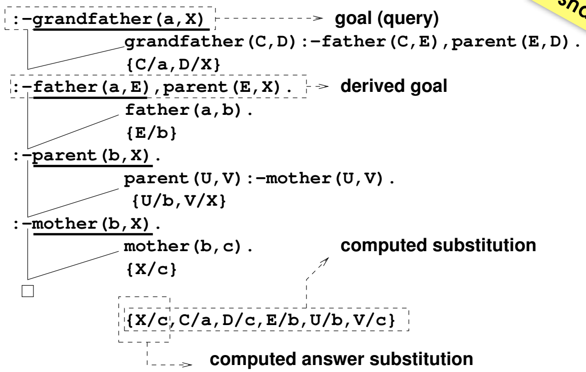
the clause obtained from a resolution step (the resolvent) is always resolved with a program clause in the next (and not with another resolvent)

refers to the shape of the resulting proof trees

SLD-resolution refutation:
refutation proof trees based on SLD-resolution

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```

linear shape!



SLD-resolution refutation:
SLD-trees

not the same as proof trees!

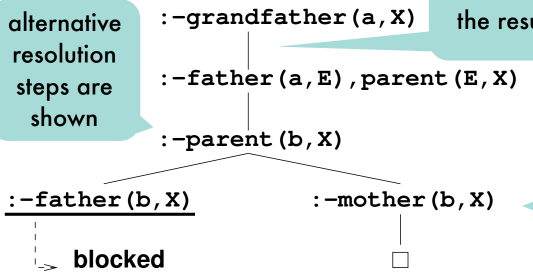
```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```

program clauses resolved with are not shown, nor are the resulting substitutions

alternative resolution steps are shown

failure branch

success branch



Prolog traverses SLD-trees depth-first, backtracking from a blocked node to the last choice point (also from a success node when more answers are requested)

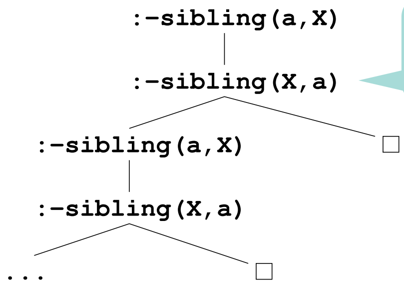
every path from the query root to the empty clause corresponds to a proof tree (a successful refutation proof)

Problems with SLD-resolution refutation:
never reaching success branch because of infinite subtrees

```
sibling(X,Y) :- sibling(Y,X).
sibling(b,a).
```

rule of thumb: non-recursive clauses before recursive ones

had we re-ordered the clauses, we would have reached a success branch at the second choice point



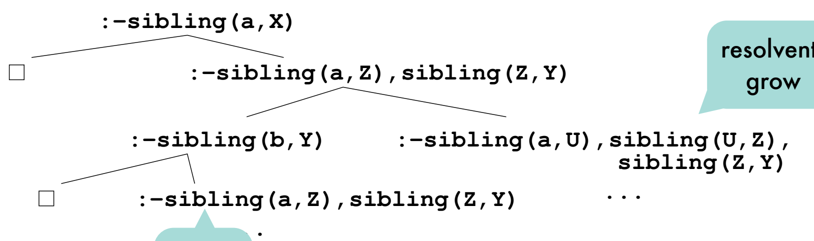
incompleteness of Prolog is a design choice: **breadth-first traversal** would require keeping all resolvents on a level in memory instead of 1

Prolog loops on this query; renders it incomplete!
only because of **depth-first traversal** and not because of resolution as all answers are represented by success branches in the SLD-tree

Problems with SLD-resolution refutation:

Prolog loops on infinite SLD-trees
when no (more) answers can be found

```
sibling(a,b).
sibling(b,c).
sibling(X,Y) :- sibling(X,Z), sibling(Z,Y).
```



resolvents grow

infinite tree

cannot be helped using breadth-first traversal: is due to **semi-decidability** of full and definite clausal logic

Problems with SLD-resolution refutation:

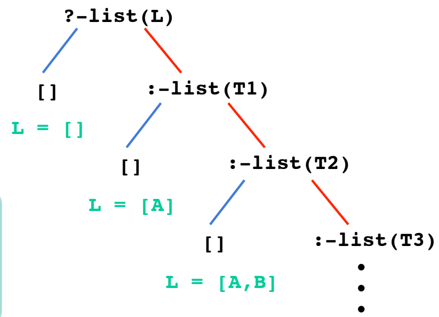
illustrated on list generation

Prolog would loop without finding answers if clauses were reversed!

```
list([]).
list([H|T]):-list(T).
```

```
?-list(L).
L = [];
L = [A];
L = [A,B];
...
```

benign:
infinitely many lists of arbitrary length are generated



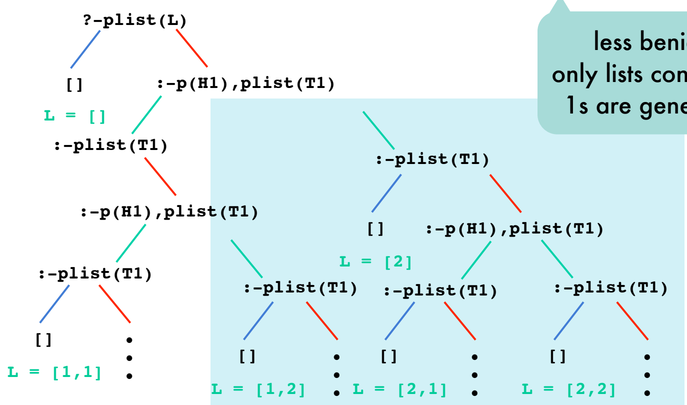
Problems with SLD-resolution refutation:

illustrated on list generation

```
plist([]).
plist([H|T]):-p(H),plist(T).
p(1).
p(2).
```

```
?-plist(L).
L=[];
L=[1];
L=[1,1];
...
```

less benign:
only lists containing 1s are generated



explored by Prolog

success branches that are never reached

SLD-resolution refutation:

implementing backtracking

amounts to going up one level in SLD-tree and descending into the next branch to the right

when a failure branch is reached (non-empty resolvent which cannot be reduced further), next alternative for the last-chosen program clause has to be tried

requires remembering previous resolvents for which not all alternatives have been explored together with the last program clause that has been explored at that point

backtracking=
popping resolvent from stack and exploring next alternative

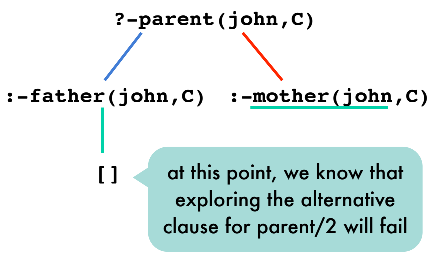
Pruning the search by means of cut: *cutting choice points*

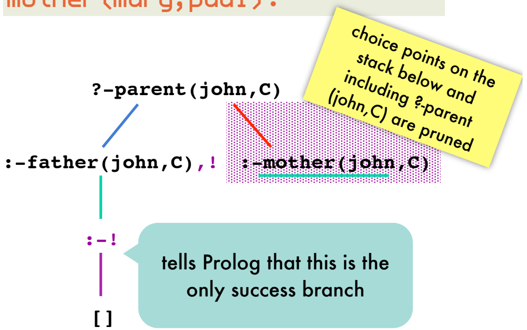
need to be **remembered** for all resolvents for which not all alternatives have been explored

unnecessary alternatives **will eventually be explored**

```
parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```

```
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```





Pruning the search by means of cut: *operational semantics*

"Once you've reached me, stick with all variable substitutions you've found after you entered my clause"

Prolog won't try alternatives for:

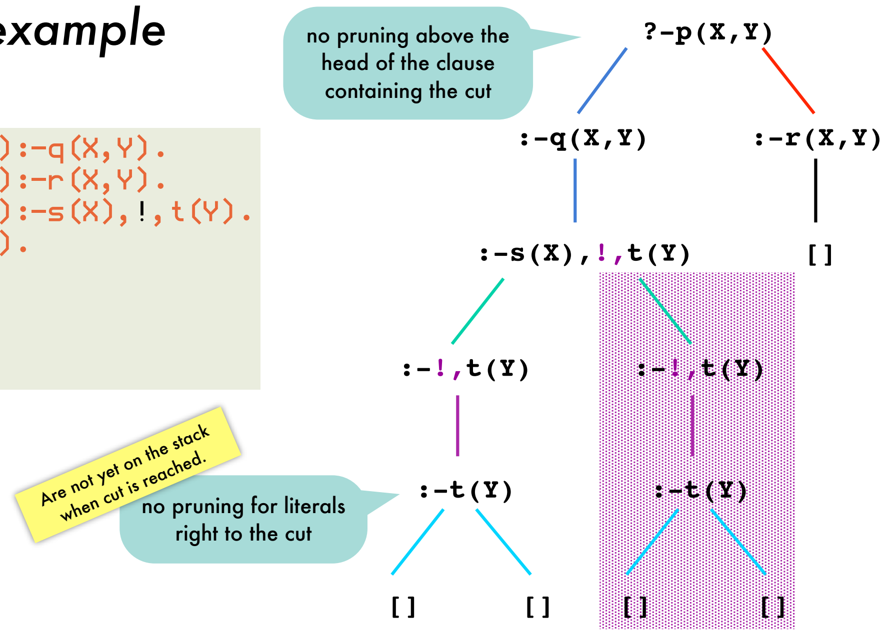
literals left to the cut

nor the clause in which the cut is found

A cut evaluates to true.

Pruning the search by means of cut: *an example*

```
p(X,Y):-q(X,Y).
p(X,Y):-r(X,Y).
q(X,Y):-s(X),!,t(Y).
r(c,d).
s(a).
s(b).
t(a).
t(b).
```



Pruning the search by means of cut: *different kinds of cut*

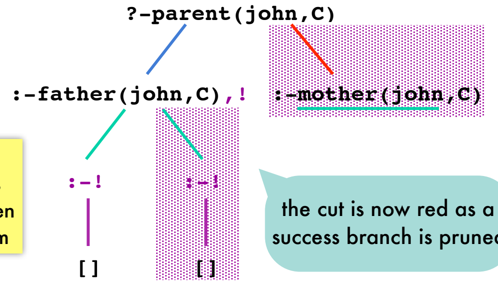
green cut	red cut
does not prune away success branches	prunes success branches
stresses that the conjuncts to its left are deterministic and therefore do not have alternative solutions	some logical consequences of the program are not returned
and that the clauses below with the same head won't result in alternative solutions either	has the declarative and procedural meaning of the program diverge

Pruning the search by means of cut: *red cuts*

```
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
father(john,peter).
mother(mary,paul).
mother(mary,peter).
```

same query, but John has multiple children in this program

{~~C/peter~~}

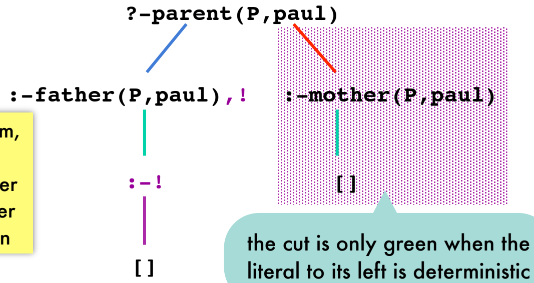


the cut is now red as a success branch is pruned

```
parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).
```

same program, but query quantifies over parents rather than children

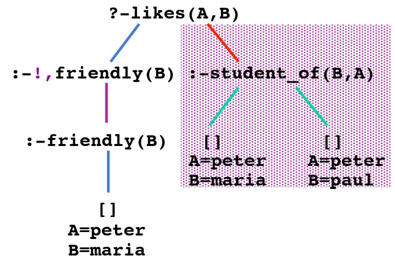
{~~P/mary~~}



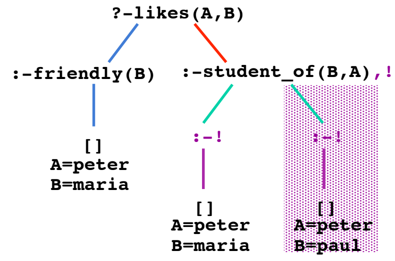
the cut is only green when the literal to its left is deterministic

Pruning the search by means of cut: *placement of cut*

```
likes(peter,Y):-friendly(Y).
likes(T,S):-student_of(S,T).
student_of(maria,peter).
student_of(paul,peter).
friendly(maria).
```



`likes(peter,Y):-!,friendly(Y).`



`likes(T,S):-student_of(S,T),!.`

Pruning the search by means of cut: *more dangers of cut*

```
max(M,N,M) :- M>=N.
max(M,N,N) :- M=<N.
```

clauses are not mutually exclusive
two ways to solve query ?-max(3,3,5)

```
max(M,N,M) :- M>=N,!.
max(M,N,N).
```

could use red cut to prune second way

only correct when used in queries with uninstantiated third argument

Better to use >= and <

problem:
?-max(5,3,3)
succeeds

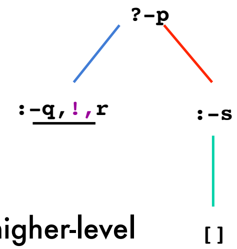
Negation as failure: *specific usage pattern of cut*

cut is often used to ensure clauses are mutually exclusive

cf. previous example

```
p :- q,!,r.
p :- s.
```

only tried when q fails



such uses are equivalent to the higher-level

```
p :- q,r.
p :- not_q,s.
```

where

```
not_q:-q,!,fail.
not_q.
```

built-in predicate always false

Prolog's not/1 meta-predicate captures such uses:

```
not(Goal) :- Goal, ! fail.
not(Goal).
```

slight abuse of syntax equivalent to call(Goal)

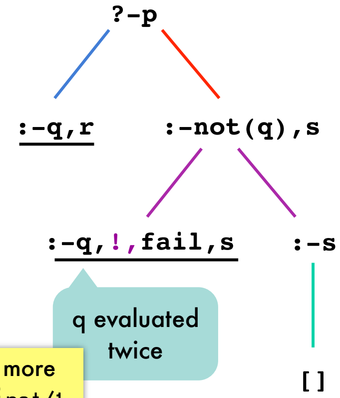
not(Goal) is proved by failing to prove Goal

in modern Prologs: use \+ instead of not

Negation as failure:

SLD-tree where not(q) succeeds because q fails

```
p:-q,r.
p:-not(q),s.
s.

not(Goal):-Goal,!,fail.
not(Goal).
```

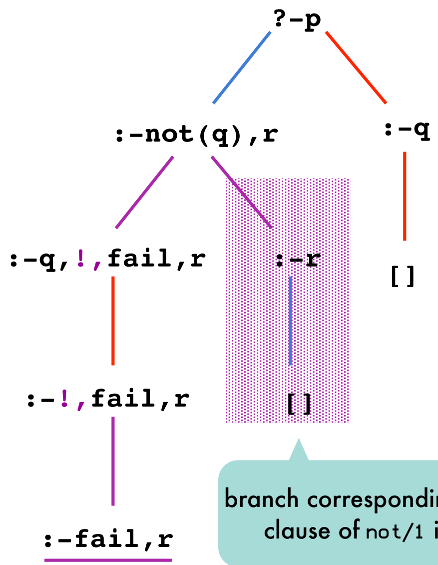


version with ! was more efficient, but uses of not/1 are easier to understand

Negation as failure:

SLD-tree where not(q) fails because q succeeds

```
p:-not(q),r.
p:-q.
q.
r.

not(Goal):-Goal,!,fail.
not(Goal).
```



branch corresponding to second clause of not/1 is pruned

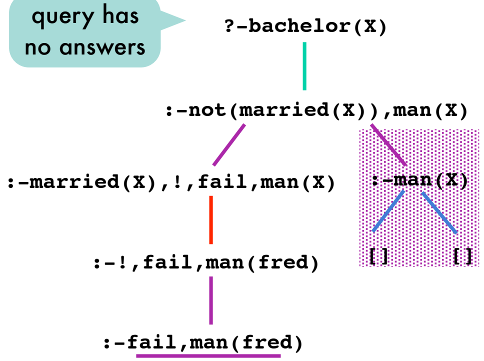
Negation as failure:

floundering occurs when argument is not ground

```
bachelor(X):-not(married(X)),
             man(X).
man(fred).
man(peter).
married(fred).
```

unintentionally interpreted as "X is a bachelor if nobody is married and X is man"

query has no answers




```
not(Goal):-Goal,!,fail.
not(Goal).
```

these are the bachelors we were looking for!

Negation as failure:

avoiding floundering

correct implementation of SLDNF-resolution:
not(Goal) fails only if Goal has a refutation with an **empty** answer substitution

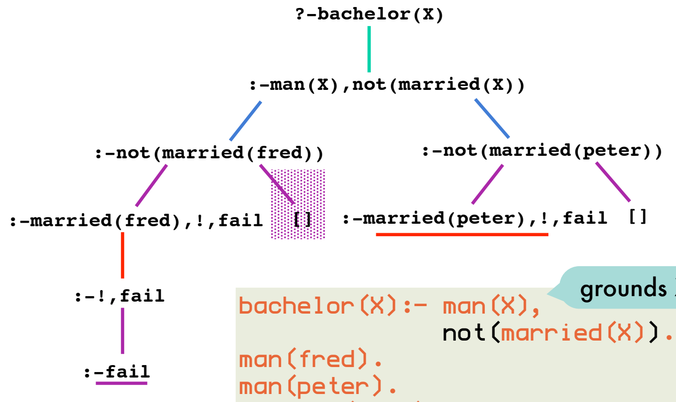
Prolog does not perform this check: not(married(X)) failed because married(X) succeeded with {X/fred}

work-around: if Goal is ground, only empty answer substitutions are possible

grounds X

```
bachelor(X):- man(X),
              not(married(X)).
man(fred).
man(peter).
married(fred).
```

Negation as failure:
avoiding floundering



grounds X

```
bachelor(X):- man(X),
              not(married(X)).
man(fred).
man(peter).
married(fred).
```


More uses of cut:
if-then-else

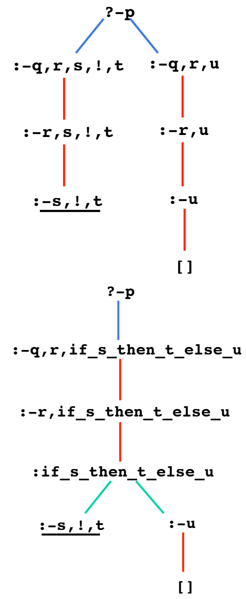
q and r evaluated twice

```
p:-q,r,s,!,t.
p:-q,r,u.
q.
r.
u.
```

only evaluated when s is false and both q and r are true

such uses are equivalent to

```
p:-q,r,if_s_then_t_else_u.
if_s_then_t_else_u:-s,!,t.
if_s_then_t_else_u:-u.
q.
r.
u.
```



More uses of cut:
if-then-else built-in

```
p :- q,r,if_then_else(S,T,U).
if_then_else(S,T,U):- S,!,T.
if_then_else(S,T,U):- U.
```

built-in as P->Q;R

nested if's:
P->Q; (R->S;T)

```
diagnosis(Patient,Condition) :-
  temperature(Patient,T),
  ( T=<37       -> blood_pressure(Patient,Condition)
  ; T>37, T<38  -> Condition=ok
  ; otherwise   -> diagnose_fever(Patient,Condition)
```

always evaluates to true

More uses of cut:
enabling tail recursion optimization

```
play(Board,Player):-
  lost(Board,Player).
play(Board,Player):-
  find_move(Board,Player,Move),
  make_move(Board,Move,NewBoard),
  next_player(Player,Next),!,
  play(NewBoard,Next).

:-play(starconfiguration,first).
```

would otherwise maintain all previous board configurations and all moves such that they can be undone

pops choice points from stack before entering next recursion

most Prolog's optimize tail recursion into iterative processes if the literals before the recursive call are deterministic

Arithmetic in Prolog: is/2

Peano-encoding of natural numbers is clumsy and inefficient

multiplication as repeated addition using recursion

```
?-X is 5+7-3.
X = 9
```

```
?-X is 5*3+7/2.
X = 18.5
```

```
?-9 is 5+7-3.
Yes
```

must be instantiated

```
?-9 is  X+7-3.
Error in arithmetic expression
```

defined as an infix operator

`is(Result,Expression)` succeeds if `Expression` can be evaluated as an arithmetic expression and its resulting value unifies with `Result`

Arithmetic in Prolog: is/2 versus =/2

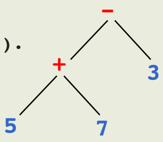
\=/2 when its arguments cannot be unified

succeeds if its arguments can be unified

```
?- X = 5+7-3
X = 5+7-3

?- 9 = 5+7-3
no

?- X = Y+3
X = _947+3
Y = _947

?- X = f(X)
X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f( ..
error: term being written is too deep
```

```
?-display(5+7-3).
-(+(5,7),3)
```

just a term

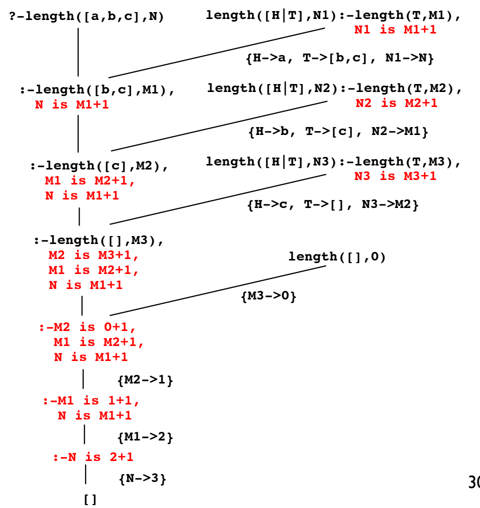


Prolog practices: *accumulators*

cannot simply place the recursive call after the is/2 literal as the latter's second argument has to be instantiated

not tail-recursive

```
length([],0).
length([H|T],N) :- length(T,N1), N is N1+1.
```



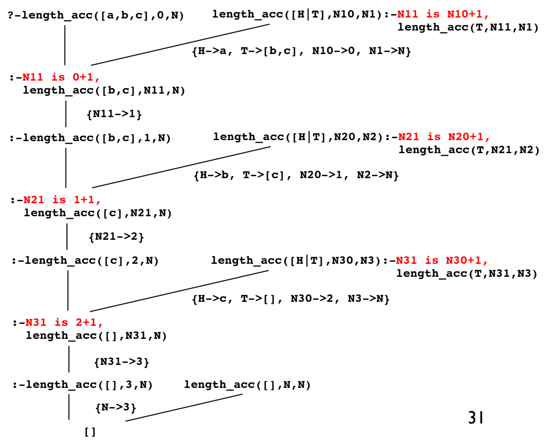
the resolvent collects as many is/2 literals as there are elements in the list before doing any actual calculation

Prolog practices: tail-recursive length/2 with accumulator

```
length(L,N) :- length_acc(L,0,N).
length_acc([],N,N).
length_acc([H|T],N0,N) :-
  N1 is N0+1,
  length_acc(T,N1,N).
```

accumulator represents length so far

read length_acc(L,M,N) as N = M + length(L)



Prolog practices: tail-recursive reverse/2 with accumulator

```
naive_reverse([],[]).
naive_reverse([H|T],R) :-
  naive_reverse(T,R1),
  append(R1,[H],R).

append([],Y,Y).
append([H|T],Y,[H|Z]) :-
  append(T,Y,Z).
```

costly

reverse(X,Y,Z)
⇔ Z=reverse(X)+Y

```
reverse(X,Z) :- reverse(X,[],Z).

reverse([],Z,Z).
reverse([H|T],Y,Z) :-
  reverse(T,[H|Y],Z).
```

reverse(X,[],Z) ⇔ Z=reverse(X)

reverse([H|T],Y,Z) ⇔ Z=reverse([H|T])+Y

⇔ Z=reverse(T)+[H]+Y

⇔ Z=reverse(T)+[H|Y]

⇔ reverse(T,[H|Y],Z)

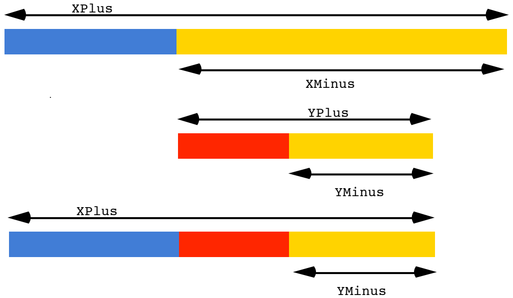
Prolog practices: difference lists

represent a list by a term L1-L2.

[a,b,c,d]-[d]	[a,b,c]
[a,b,c,1,2]-[1,2]	[a,b,c]
[a,b,c|X]-X	[a,b,c]

variable for minus list:
can be used as pointer to end of represented list

Prolog practices: appending difference lists in constant time



one unification step rather than as many resolution steps as there are elements in the list appended to

```
append_dl(XPlus-XMinus,YPlus-YMinus,XPlus-YMinus) :- XMinus=YPlus.
```

or

```
append_dl(XPlus-YPlus,YPlus-YMinus,XPlus-YMinus).

?-append_dl([a,b|X]-X, [c,d|Y]-Y,Z).
X = [c,d|Y], Z = [a,b,c,d|Y]-Y
```


Prolog practices: reversing difference lists

reverse(X,Y,Z) ⇔ Z=reverse(X)+Y

⇔ reverse(X)=Z-Y

reverse([H|T],Y,Z) ⇔ Z=reverse([H|T])+Y

⇔ Z=reverse(T)+[H|Y]

⇔ reverse(T)=Z-[H|Y]

```
reverse(X,Z) :- reverse_dl(X,Z-[]).

reverse_dl([],Z-Z).
reverse_dl([H|T],Z-Y) :- reverse_dl(T,Z-[H|Y]).
```

Second-order predicates: map/3

```
map(R, [], []).
map(R, [X|Xs], [Y|Ys]):-R(X,Y),map(R,Xs,Ys).
?-map(parent, [a,b,c],X)
```

or, when atoms with variable as predicate symbol are not allowed:

```
map(R, [], []).
map(R, [X|Xs], [Y|Ys]):- Goal =.. [R,X,Y],
                         call(Goal),
                         map(R,Xs,Ys).
```

univ operator =.. can be used to construct terms:
?-Term=..[parent,X,peter]
Term=parent(X,peter)

and decompose terms:
?-parent(maria,Y)=..List
List=[parent,maria,Y]

Term=..List succeeds
if Term is a constant and List is the list [Term]
if Term is a compound term f(A1,..,An)
and List is a list with head f and whose tail unifies with [A1,..,An]

Second-order predicates: findall/3

findall(Template,Goal,List) succeeds if List unifies with a list of the terms Template is instantiated to successively on backtracking over Goal. If Goal has no solutions, List has to unify with the empty list.

```
parent(john,peter).
parent(john,paul).
parent(john,mary).
parent(mick,davy).
parent(mick,dee).
parent(mick,dozy).
```

```
?-findall(C,parent(john,C),L).
  L = [peter,paul,mary]

?-findall(f(C),parent(john,C),L).
  L = [f(peter),f(paul),f(mary)]

?-findall(C,parent(P,C),L).
  L = [peter,paul,mary,davy,dee,dozy]
```


Second-order predicates: bagof/3 and setof/3

differ from findall/3 if Goal contains free variables

```
parent(john,peter).
parent(john,paul).
parent(john,mary).
parent(mick,davy).
parent(mick,dee).
parent(mick,dozy).
```

```
?-findall(C,parent(P,C),L).
  L = [peter,paul,mary,davy,dee,dozy]

?-bagof(C,parent(P,C),L).
  P = john
  L = [peter,paul,mary];

  P = mick
  L = [davy,dee,dozy]

?-bagof(C,P^parent(P,C),L).
  L = [peter,paul,mary,davy,dee,dozy]
```

a parent and its list of children

list of children for which a parent exists

The construct *Var^Goal* tells bagof/3 not to bind *Var* in *Goal*.

setof/3 is same as bagof/3 without duplicate elements in List

findall/3 is same as bagof/3 with all free variables existentially quantified using ^

Second-order predicates: assert/1 and retract/1

Backtracking over such literals will not undo the modifications to the database!

asserta(Clause)
adds Clause at the beginning of the Prolog database.
assertz(Clause) and assert(Clause)
adds Clause at the end of the Prolog database.
retract(Clause)
removes first clause that unifies with Clause from the Prolog database.

retract all clauses of which the head unifies with Term

```
retractall(Term):-
  retract(Term), fail.
retractall(Term):-
  retract((Term:- Body)), fail.
retractall(Term).
```

failure-driven loop

Second-order predicates: assert/1 and retract/1

Powerful: enable run-time program modification
Harmful: code hard to understand and debug, often slow

sometimes used as global variables, “boolean” flags or to memoize:

```
fib(0,0).
fib(1,1).
fib(N,F) :-
  N > 1,
  N1 is N-1,
  N2 is N1-1,
  fib(N1,F1),
  fib(N2,F2),
  F is F1+F2.
```

```
mfib(N, F):- memo_fib(N, F), !.
mfib(N, F):-
  N > 1,
  N1 is N-1,
  N2 is N1-1,
  mfib(N1,F1),
  mfib(N2,F2),
  F is F1+F2,
  assert(memo_fib(N, F)).

:- dynamic memo_fib/2.
memo_fib(0,0).
memo_fib(1,1).
```

if you’ve remembered an answer for this goal before, return it

most Prologs require such a declaration for clauses that are added or removed from the program at run-time

[Slides on Computational Logic from CLIP group]

Higher-order programming using call/N: call(Goal,...)

a more flexible form of call/1, which takes additional arguments that will be added to the Goal that is called

```
call(p(X1,X2,X3))
call(p(X1,X2), X3)
call(p(X1), X2, X3)
call(p, X1, X2, X3)
```

all result in `p(X1, X2, X3)` being called

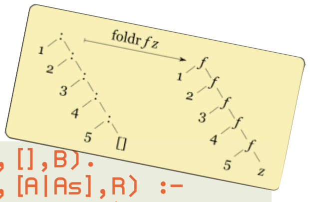
Supported by most Prolog systems in addition to call/1

can often be used in places where you would use univ operator =.. to construct the goal

[Higher-order logic programming in Prolog, Lee Naish, 1996]

Higher-order programming using call/N: implementing map and friends

```
map(_F, [], []).
map(F, [A0|As0], [A|As]) :-
  call(F,A0,A),
  map(F,As0,As).
```




```
foldr(F,B, [],B).
foldr(F,B, [A|As],R) :-
  foldr(F,B,As,R1),
  call(F,A,R1,R).
```

```
filter(_P, [], []).
filter(P, [A0|As0],As) :-
  (call(P, A0) ->
    As = [A0|As1]
   ;As = As1),
  filter(P, As0, As1)
```

```
compose(F,G,X,FGX):-
  call(G,X,GX),
  call(F,GX,FGX).
```

[Higher-order logic programming in Prolog, Lee Naish, 1996]

Higher-order programming using call/N: using map and friends (1)

```
?- filter(>(5), [3,4,5,6,7],As).
As= [3,4]

?- map(plus(1), [2,3,4],As).
As= [3,4,5]

?- map(between(1), [2,3],As).
As= [1,1]; As= [1,2]; As= [1,3];
As= [2,1]; As= [2,2]; As= [2,3]

?- map(plus(1),As, [3,4,5]).
As= [2,3,4]

?- map(plus(X), [2,3,4], [3,4,5]).
X=1

?- map(plus(X), [2,A,4], [3,4,B]).
X=1,A=3,B=5
```

called goal: >(5,X)

between(I,J,X) binds X to an integer between I and J inclusive.

assuming that plus/3 is reversible (e.g., Peano arithmetic)

relies on execution order in which X is bound first

[Higher-order logic programming in Prolog, Lee Naish, 1996]

Higher-order programming using call/N: using map and friends (2)

flatten defined in terms of foldr using empty list and append

```
?- foldr(append, [], [[2], [3,4], [5]],As).
As=[2,3,4,5]

?- compose(map(plus(1)),foldr(append, []), [[2], [3,4], [5]],As).
As=[3,4,5,6]
```

flattens first, then adds 1

plain Prolog lacks "currying" for higher-order programming: functional programming languages would return a list of functions that take the missing argument

conceptual difficulty: ok to curry a call(sum(2,3)) to a sum(2,3,Z) if there is also a definition for sum(X,Y)?

```
?- map(plus, [2, 3, 4], As).
ERROR: map/3: Undefined procedure: plus/2
ERROR:     However, there are definitions for:
ERROR:         plus/3
```

[Higher-order logic programming in Prolog, Lee Naish, 1996]

Inspecting terms: var/1 and its use in practice

var(Term)
succeeds when Term is an uninstantiated variable
nonvar(Term) has opposite behavior

```
?- var(X).
true.
?- X=3,var(X).
false.
```

ensuring relational nature of predicates

```
plus(X,Y,Z) :-
  nonvar(X),nonvar(Y),Z is X+Y.
plus(X,Y,Z) :-
  nonvar(X),nonvar(Z),Y is Z-X.
plus(X,Y,Z) :-
  nonvar(Y),nonvar(Z),X is Z-Y.
```

directing search for efficiency

```
grandparent(X,Z) :-
   nonvar(X),parent(X,Y),parent(Y,Z).
grandparent(X,Z) :-
   nonvar(Z),parent(Y,Z),parent(X,Y).
```


Inspecting terms: arg/3 and functor/3

complement =.. operator

arg(N,Term,Arg)
succeeds when Arg is the Nth argument of Term
functor(Term,F,N)
succeeds when the Term starts with the functor F of arity N

tests whether a term is ground (i.e., contains no uninstantiated variables)

```
ground(Term) :-
  nonvar(Term),constant(Term).
ground(Term) :-
  nonvar(Term),
  compound(Term),
  functor(Term,F,N),
  ground(N,Term).
ground(N,Term) :-
   N > 0,
   arg(N,Term,Arg),
   ground(Arg),
   N1 is N-1,
   ground(N1,Term).
ground(0,Term).
```

common Prolog practice: arity of auxiliary and main predicates differ

Extending Prolog: term_expansion(+In,-Out)

called by Prolog for each file it compiles

clause or list of clauses that will be added to the program instead of the In clause

useful for generation code, e.g. :

given compound term representation of data

```
student(Name,Id)
```

want to use accessor predicates

```
student_name(student(Name, _), Name).
student_id(student(_, Id), Id).
```

instead of explicit unifications throughout the code

```
Student = student(Name,_)
```

to ensure independence of one particular representation of the data

[http://ww2.cs.mu.oz.au/255/last_semester/lec/subject-prolog_meta.pdf]

Extending Prolog: term_expansion(+In,-Out)

```
:- struct student(name,id).
```

↓

```
student_name(student(Name, _), Name).
student_id(student(_, Id), Id).
```

declares struct as a prefix operator

```
:- op(1150, fx, (struct)).

term_expansion((:- struct Term), Clauses) :-
   functor(Term, Name, Arity),
   functor(Template, Name, Arity),
   gen_clauses(Arity, Name, Term, Template, Clauses).
```

create Template with same functor and arity, but with variable arguments rather than constants

[http://ww2.cs.mu. … t_semester/lec/subject-prolog_meta.pdf]

Extending Prolog: term_expansion(+In,-Out)

```
?- X=0'_.
X = 95.
?- char_code(X,95).
X = '_'.
```

N-th argument recursed upon

trick to merge recursive and base clause

conversion from atom to list of character codes

creates fact

```
gen_clauses(N, Name, Term, Template, Clauses) :-
 (N =:= 0 ->
   Clauses = []
 ;arg(N, Term, Argname),
  arg(N, Template, Arg),
  atom_codes(Argname, Argcodes),
  atom_codes(Name, Namecodes),
  append(Namecodes, [0'_|Argcodes],Codes),
  atom_codes(Pred, Codes),
  Clause =.. [Pred, Template, Arg],
  Clauses = [Clause|Clauses1],
  N1 is N - 1,
  gen_clauses(N1, Name, Term, Template, Clauses1)
 ).
```

When trying out, put gen_clauses/5 before term_expansion/2

[http://ww2.cs.mu.oz.au/255/last_semester/lec/subject-prolog_me…

Extending Prolog: operators

Certain functors and predicate symbols that be used in infix, prefix, or postfix rather than term notation.

```
:- op(500,xfx,'has_color').
a has_color red.
b has_color blue.
```

```
?- b has_color C.
C = blue.
?- What has_color red.
What = a
```

integer between 1 and 1200;
smaller integer binds stronger
a+b/c ≡ a+(b/c) ≡ +(a,/(b,c)) if / smaller than +

```
:- op(Precedence, Type, Name)
```

prefix: fx, fy
infix: xfx, xfy,yfx
postfix: xf,yf

associative	not	right	left
	xfx	xfy	yfx
X op Y op Z	/	op(X,op(Y,Z))	op(op(X,Y),Z)

Extending Prolog: operators in towers of Hanoi

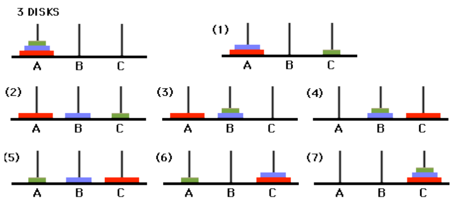
Moves is the list of moves to move N discs from peg A to peg C, using peg B as an intermediary.

```
:- op(900,xfx,to).
hanoi(0,A,B,C,[]).
hanoi(N,A,B,C,Moves):-
  N1 is N-1,
  hanoi(N1,A,C,B,Moves1),
  hanoi(N1,B,A,C,Moves2),
  append(Moves1,[A to C|Moves2],Moves).
```

move n-1 c from A to B. disc #n is left on A

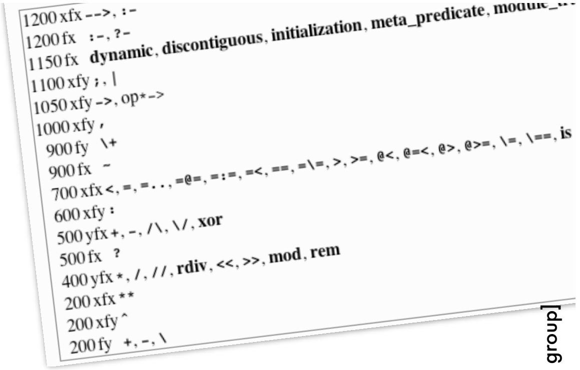
move n-1 discs from B to C. they will rest on disc #n

move disc #n from A to C




```
?- hanoi(3,left,middle,right,M)
M = [left to right,
     left to middle,
     right to middle,
     left to right,
     middle to left,
     middle to right,
     left to right ]
```

Extending Prolog: built-in operators



1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	dynamic, discontiguous, initialization, meta_predicate, module_...
1100	xfy	;, |
1050	xfy	->, op*->
1000	xfy	,
900	fy	\+
900	fx	~
700	xfx	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is
600	xfy	:
500	yfx	+, -, /\, \/, xor
500	fx	?
400	yfx	*, /, //, rdiv, <<, >>, mod, rem
200	xfx	**
200	xfy	^
200	fy	+, -, \

```
+'(a,'/'(b,c))              a+b/c
is(X, mod(34, 7))           X is 34 mod 7
<'('+'(3,4),8)              3+4<8
'='(X,f(Y))                 X=f(Y)
'-'(3)                      -3
':-'(p(X),q(Y))             p(X) :- q(Y)
':-'(p(X),','(q(Y),r(Z)))   p(X) :- q(Y),r(Z)
```

clauses are also Prolog terms!

[Slides on Computational Logic from CLIP group]

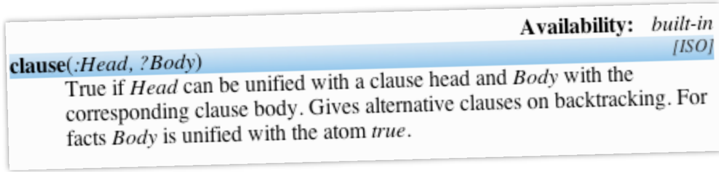
Extending Prolog: vanilla and canonical naf meta-interpreter

```
prove(Goal):-
  clause(Goal,Body),
  prove(Body).

prove((Goal1,Goal2)):-
  prove(Goal1),
  prove(Goal2).

prove(true).
```

Are these meta-circular interpreters?

```
prove(true):- !.

prove((A,B)):- !,
  prove(A),
  prove(B).

prove(not(Goal)):- !,
  not(prove(Goal)).

prove(A):-
  % not (A=true; A=(X,Y); A=not(G))
  clause(A,B),
  prove(B).
```

Avoids problems where clause/2 is called with a conjunction or true.



Availability: *built-in*

clause(*:Head, ?Body*) *[ISO]*

True if *Head* can be unified with a clause head and *Body* with the corresponding clause body. Gives alternative clauses on backtracking. For facts *Body* is unified with the atom *true*.

Extending Prolog: meta-level vs object-level in meta-interpreter

	KNOWLEDGE	REASONING
META-LEVEL	`clause(p(X),q(X)).` `clause(q(a),true).`	`?-prove(p(X)).` `X=a`
OBJECT-LEVEL	`p(X):-q(X).` `q(a).`	`?-p(X).` `X=a`

Canonical meta-interpreter still **absorbs** backtracking, unification and variable environments implicitly from the object-level.

Reified unification explicit at meta-level :

```
prove(A):-
  clause(Head,Body),
  unify(A,Head,MGU,Result),
  apply(Body,MGU,NewBody),
  prove_var(NewBody).
```


Prolog programming: a methodology illustrated on partition/4

(might not work equally well for everyone)

1 Write down declarative specification

```
% partition(L,N,Littles,Bigs) <- Littles contains numbers
%                                in L smaller than N,
%                                Bigs contains the rest
```

2 Identify recursion and "output" arguments

what is the recursion argument?
what is the base case?

3 Write down implementation skeleton

```
partition([],N,[],[]).
partition([Head|Tail],N,?Littles,?Bigs):-
  /* do something with Head */
  partition(Tail,N,Littles,Bigs).
```

Empty list is partitioned into two empty lists.

We recurse on the "input" argument list.

Prolog programming: a methodology illustrated on partition/4

4 Complete bodies of clauses

```
partition([],N, [], []).
partition([Head|Tail],N,?Littles,?Bigs):-
  Head < N,
  partition(Tail,N,Littles,Bigs),
  ?Littles = [Head|Littles],?Bigs = Bigs.
partition([Head|Tail],N,?Littles,?Bigs):-
  Head >= N,
  partition(Tail,N,Littles,Bigs),
  ?Littles = Littles,?Bigs = [Head|Bigs].
```

Head is smaller, has to be added to Littles

has to be added to Bigs otherwise

5 Fill in "output" arguments

```
partition([],N, [], []).
partition([Head|Tail],N, [Head|Littles],Bigs):-
  Head < N,
  partition(Tail,N,Littles,Bigs).
partition([Head|Tail],N,Littles, [Head|Bigs]):-
  Head >= N,
  partition(Tail,N,Littles,Bigs).
```


Prolog programming: a methodology illustrated on sort/2

1 Write down declarative specification

```
% sort(L,S) <- S is a sorted permutation of list L
```

2 Identify recursion and "output" arguments

3 Write down implementation skeleton

```
sort([], []).
sort([Head|Tail],?Sorted):-
  /* do something with Head */
  sort(Tail,Sorted).
```

4 Complete bodies of clauses

```
sort([], []).
sort([Head|Tail],WholeSorted):-
  sort(Tail,Sorted),
  insert(Head,Sorted,WholeSorted).
```

Auxiliary predicate

Prolog programming: a methodology illustrated on insert/3

1 Write down declarative specification

```
% insert(X,In,Out) <- In is a sorted list, Out is In
%                     with X inserted in the proper place
```

2 Identify recursion and "output" arguments

3 Write down implementation skeleton

```
insert(X, [],?Inserted).
insert(X, [Head|Tail],?Inserted):-
  /* do something with Head */
  insert(X,Tail,Inserted).
```


Prolog programming: a methodology illustrated on insert/3

4 Complete bodies of clauses

```
insert(X, [],?Inserted):-
  ?Inserted= [X].
insert(X, [Head|Tail],?Inserted):-
  X > Head,
  insert(X,Tail,Inserted),
  ?Inserted = [Head|Inserted].
insert(X, [Head|Tail],?Inserted):-
  X =< Head,
  ?Inserted = [X,Head|Tail].
```

5 Fill in "output" arguments

```
insert(X, [], [X]).
insert(X, [Head|Tail], [X,Head|Tail]):-
  X =< Head.
insert(X, [Head|Tail], [Head|Inserted]):-
  X > Head,
  insert(X,Tail,Inserted).
```

More Prolog programming: quicksort

```
quicksort([], []).
quicksort([X|Xs],Sorted):-
  partition(Xs,X,Littles,Bigs),
  quicksort(Littles,SortedLittles),
  quicksort(Bigs,SortedBigs),
  append(SortedLittles, [X|SortedBigs],Sorted).
```

with difference lists:

```
quicksort(Xs,Ys) :- qsort(Xs,Ys-[]).

qsort([],Ys-Ys).
qsort([X0|Xs],Ys-Zs) :-
  partition(Xs,X0,Ls,Bs),
  qsort(Bs,Ys2-Zs),
  qsort(Ls,Ys-[X0|Ys2]).
```


Revisiting the Eliza classic in Prolog: example conversation

„I am unhappy."
„How long have you been unhappy?"
„Six months. Can you help me?"
„What makes you think I help you?"
„You remind me of my sister."
„Can you tell me more about your sister?"
„I like teasing my sister."
„Does anyone else in your family like teasing your sister?"
„No. Only me."
„Please go on."

[The Art of Prolog, Sterling and Shapiro]

Revisiting the Eliza classic in Prolog: core "algorithm"

driven by stimulus-response patterns

I am <statement>.
How long have you been <statement>?

while the input is not „bye"
- choose a stimulus-response pair
- match the input to the stimulus
- generate the reply from the response and the match
- output the response

[The Art of Prolog, Sterling and Shapiro]

Revisiting the Eliza classic in Prolog: dictionary lookup

as association list for arbitrary keys:

```
lookup(Key, [(Key,Value)|Dict],Value).
lookup(Key, [(Key1,Value1)|Dict],Value) :-
  Key \= Key1,
  lookup(Key,Dict,Value).
```

will be used to store matches between stimulus and input

as binary tree for integer keys:

```
lookup2(Key,dict(Key,X,Left,Right),Value) :- !,
  X = Value.
lookup2(Key,dict(Key1,X,Left,Right),Value) :-
  Key < Key1,
  lookup2(Key,Left,Value).
lookup2(Key,dict(Key1,X,Left,Right),Value) :-
  Key > Key1,
  lookup2(Key,Right,Value).
```

[The Art of Prolog, Sterling and Shapiro]

Revisiting the Eliza classic in Prolog: representing stimulus/response patterns

numbered place-holder

numbered place-holder

```
pattern([i,am,1], ['How',long,have,you,been,1,?]).
pattern([1,you,2,me], ['What',makes,you,think,'I',2,you,?]).
pattern([i,like,1], ['Does',anyone,else,in,your,family,like,1,?]).
pattern([i,feel,1], ['Do',you,often,feel,that,way,?]).
pattern([1,X,2], ['Please',you,tell,me,more,about,X]) :-
  important(X).
pattern([1], ['Please',go,on,'.']).

important(father).
important(mother).
important(sister).
important(brother).
important(son).
important(daughter).
```

conditional pattern

[The Art of Prolog, Sterling and Shapiro]

Revisiting the Eliza classic in Prolog: main loop

```
reply([]) :- nl.
reply([Head|Tail]) :- write(Head),write(' '),reply(Tail).
```

```
eliza :- read(Input),
         eliza(Input),
         !.
eliza([bye]) :-
  writeln(['Goodbye. I hope I have helped you']).
eliza(Input) :-
  pattern(Stimulus,Response),
  match(Stimulus,Table,Input),
  match(Response,Table,Output),
  reply(Output),
  read(Input1),
  !,
  eliza(Input1).
```

find a Stimulus

match it with the Input, storing matches for place-holders in Table

substitute place-holders in Output

[The Art of Prolog, Sterling and Shapiro]

Revisiting the Eliza classic in Prolog: actual matching

```
match([N|Pattern],Table,Target) :-
  integer(N),
  lookup(N,Table,LeftTarget),
  append(LeftTarget,RightTarget,Target),
  match(Pattern,Table,RightTarget).
match([Word|Pattern],Table, [Word|Target]) :-
  atom(Word),
  match(Pattern,Table,Target).
match([],Table, []).
```

place-holder

word

suppose D = [(a,b),(c,d)|X]

```
?- lookup(a,D,V)
V=b
?- lookup(c,D,e)
no
?- lookup(e,D,f)
yes
% D = [(a,b),(c,d),(e,f)|X]
```

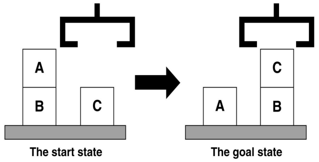
The incomplete datastructure does not have to be initialized!

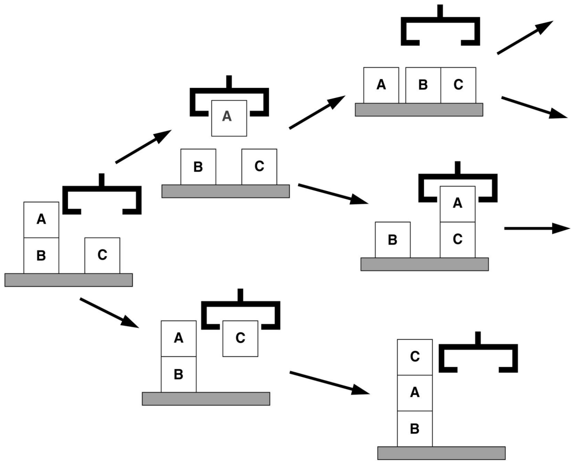
[The Art of Prolog, Sterling and Shapiro]

Declarative Programming

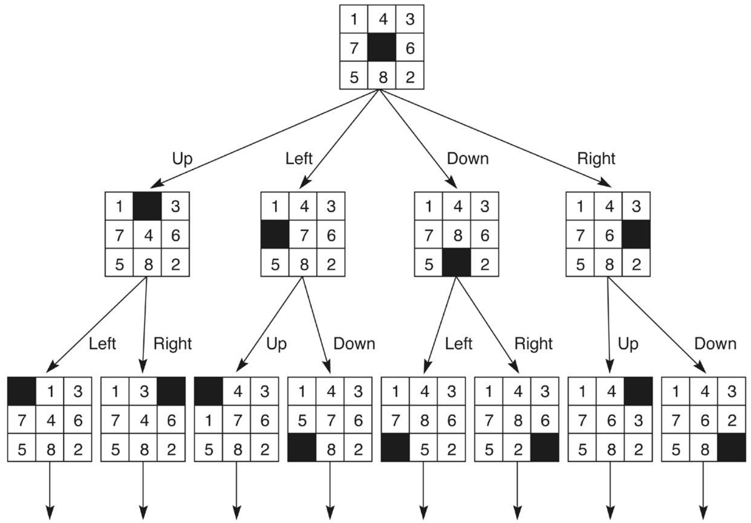
4: blind and informed search of state space, proving as search process

State space search:
blocks world





State space search:
8-puzzle



State space search:
graph representation

state space

state=node, state transition=arc

goal nodes and start nodes

cost associated with arcs between nodes

solution

path from start to goal node

optimal if cost over path is minimal

search algorithms

completeness: will a solution always be found if there is one?

optimality: will highest-quality solution be found when there are several?

efficiency: runtime and memory requirements

blind vs informed: does quality of partial solutions steer process?

State space search:
Prolog skeleton for search algorithms

```
search(Agenda,Goal):-
  next(Agenda,Goal,Rest),
  goal(Goal).

search(Agenda,Goal):-
  next(Agenda,Current,Rest),
  children(Current,Children),
  add(Children,Rest,NewAgenda),
  search(NewAgenda,Goal).
```

reached, but untested states

goal state for which goal(Goal) succeeds

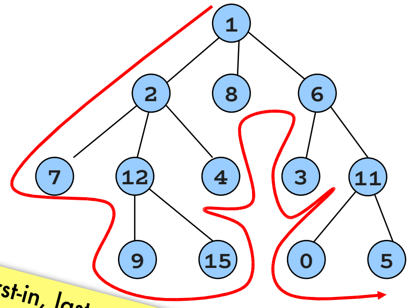
succeeds if the goal state Goal can be reached from a state on the Agenda

selects a candidate state from the Agenda

expands the current state

State space search: *depth-first search*

```
arc(1,2). arc(1,8). arc(1,6).
arc(2,7). arc(2,12). arc(2,4).
arc(12,9). arc(12,15). arc(6,3).
arc(6,11). arc(11,0). arc(11,5).
```



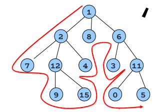
next/3 implemented by taking first element of list

first-in, last-out agenda treated as a stack

```
search_df([Goal|Rest],Goal):-
  goal(Goal).
search_df([Current|Rest],Goal):-
  children(Current,Children),
  append(Children,Rest,NewAgenda),
  search_df(NewAgenda,Goal).
children(Node,Children):-
  findall(C,arc(Node,C),Children).
```

add/3 implemented by prepending children of first element on agenda to the remainder of the agenda

State space search: *depth-first search with paths*

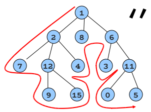


keep path to node on agenda, rather than node

only requires a change to children/3 AND way search_df/2 is called

```
children([Node|RestOfPath],Children):-
  findall([Child,Node|RestOfPath],arc(Node,Child),Children).

?- search_df([[initial_node]],PathToGoal).
```


State space search: *depth-first search with loop detection*



keep list of visited nodes

```
search_df_loop([Goal|Rest],Visited,Goal):-
  goal(Goal).
search_df_loop([Current|Rest],Visited,Goal):-
  children(Current,Children),
  add_df(Children,Rest,Visited,NewAgenda),
  search_df_loop(NewAgenda, [Current|Visited],Goal).
```

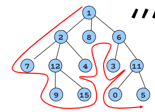
add current node to list of visited nodes

```
add_df([],Agenda,Visited,Agenda).
add_df([Child|Rest],OldAgenda,Visited, [Child|NewAgenda]):-
  not(element(Child,OldAgenda)),
  not(element(Child,Visited)),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
  element(Child,OldAgenda),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
add_df([Child|Rest],OldAgenda,Visited,NewAgenda):-
  element(Child,Visited),
  add_df(Rest,OldAgenda,Visited,NewAgenda).
```

do not add a child if it's already on the agenda

do not add already visited children

State space search: *depth-first search using Prolog stack*



vanilla

```
search_df(Goal,Goal):-
  goal(Goal).
search_df(CurrentNode,Goal):-
  arc(CurrentNode,Child),
  search_df(Child,Goal).
```

use Prolog call stack as agenda

might loop on cycles

depth bounded

```
search_bd(Depth,Goal,Goal):-
  goal(Goal).
search_bd(Depth, CurrentNode, Goal):-
  Depth>0,
  NewDepth is Depth-1,
  arc(CurrentNode, Child),
  search_bd(NewDepth, Child, Goal).

?- search_df(10,initial_node,Goal).
```

do not exceed depth threshold while searching

always halts, but no solutions beyond threshold

iterative deepening

```
search_id(CurrentNode, Goal):-
  search_id(1,CurrentNode,Goal).
search_id(Depth,CurrentNode,Goal):-
  search_bd(Depth,CurrentNode,Goal).
search_id(Depth,CurrentNode,Goal):-
  NewDepth is Depth+1,
  search_id(NewDepth,CurrentNode,Goal)
```

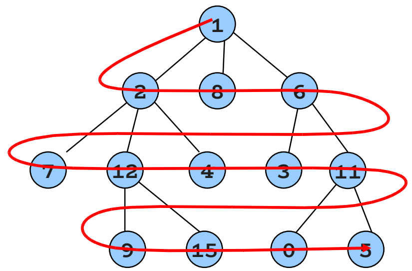
less memory than bfs

increase depth bound on each iteration

complete and solutions on, but upper parts of search space

not that bad for full trees: number of nodes at a single level is smaller than all nodes above it

State space search:
breadth-first search



next/3 implemented by taking first element of list

first-in, first-out agenda treated as a queue

```
search_bf([Goal|Rest],Goal):-
  goal(Goal).
search_bf([Current|Rest],Goal):-
  children(Current,Children),
  append(Rest,Children,NewAgenda),
  search_bf(NewAgenda,Goal).

children(Node,Children):-
  findall(C,arc(Node,C),Children).
```

add/3 implemented by appending children of first element on agenda to the remainder of the agenda

State space search:
dfs vs bfs

l=depth-limit
b=branching factor of search space
d=depth of search space
m=depth of shortest path solution

spirals away from start node, # candidate paths to be remembered grows exponentially with depth

	breadth-first	depth-first	depth-limited	iterative deepening
time	b^d	b^m	b^l	b^d
space	b^d	bm	bl	bd
shortest solution path	√			√
complete	√		√ if l≥d	√

might be second child of root node

State space search:
water jugs problem

20L 5L 8L

[The Art of Prolog, Sterling and Shapiro]

operations

- fill a jug from the pool
- empty a jug into the pool
- pour one jug into another until one poured from is empty or the one poured into is full

goal

4L in a jug

State space search:
implementing the search

[The Art of Prolog, Sterling and Shapiro]

as a generic algorithm for state space problems

visited states

sequence of transitions to reach goal from current state

until now, we only had unnamed arcs

multiple named transitions out of a state

```
solve_dfs(State,History, []) :-
  final_state(State).
solve_dfs(State,History, [Move|Moves]) :-
  move(State,Move),
  update(State,Move,State1),
  legal(State1),
  not(member(State1,History)),
  solve_dfs(State1, [State1|History],Moves).

test_dfs(Problem,Moves) :-
  initial_state(Problem,State),
  solve_dfs(State, [State],Moves).
```

State space search:
encoding water jugs problem

[The Art of Prolog, Sterling and Shapiro]

starting and goal states

```
initial_state(jugs,jugs(0,0)).
final_state(jugs(4,V2)).
final_state(jugs(V1,4)).
```

possible transitions out of a state

```
move(jugs(V1,V2),fill(1)).
move(jugs(V1,V2),fill(2)).
move(jugs(V1,V2),empty(1)) :- V1>0.
move(jugs(V1,V2),empty(2)) :- V2>0.
move(jugs(V1,V2),transfer(2,1)).
move(jugs(V1,V2),transfer(1,2)).
```

empty first jug (1), but only if it still contains water (C1)

State space search:
encoding water jugs problem

[The Art of Prolog, Sterling and Shapiro]

states a transition can lead to

```
update(jugs(V1,V2),fill(1),jugs(C1,V2)) :-
  capacity(1,C1).
update(jugs(V1,V2),fill(2),jugs(V1,C2)) :-
  capacity(2,C2).
update(jugs(V1,V2),empty(1),jugs(0,V2)).
update(jugs(V1,V2),empty(2),jugs(V1,0)).
update(jugs(V1,V2),transfer(2,1),jugs(W1,W2)) :-
  capacity(1,C1),
  Liquid is V1 + V2,
  Excess is Liquid - C1,
  adjust(Liquid,Excess,W1,W2).
update(jugs(V1,V2),transfer(1,2),jugs(W1,W2)) :-
  capacity(2,C2),
  Liquid is V1 + V2,
  Excess is Liquid - C2,
  adjust(Liquid,Excess,W2,W1).
```

a jug can be filled up to its capacity from the pool

the first jug will contain 0L after emptying it

the first jug can be poured in the second

```
adjust(Liquid, Excess,Liquid,0) :- Excess =< 0.
adjust(Liquid,Excess,V,Excess) :-
  Excess > 0,
  V is Liquid - Excess.
```

```
capacity(j1,8).
capacity(j2,5).
legal(jugs(C1,C2)).
```


Proving as a search process:
df agenda-based meta-interpreter

true: empty conjunctions
single term: singleton conjunction

```
prove(true):- !.
prove((A,B)):-
  !,
  clause(A,C),
  conj_append(C,B,D),
  prove(D).
prove(A):-
  clause(A,B),
  prove(B).
```

instead of prove((A,B)) :- prove(A),prove(B)

```
conj_append(true,Ys,Ys).
conj_append(X,Ys,(X,Ys)):-
  not(X=true),
  not(X=(One,TheOther)).
conj_append((X,Xs),Ys,(X,Zs)):-
  conj_append(Xs,Ys,Zs).
```

depth-first

```
prove_df_a(Goal) :-
  prove_df_a([Goal]).
prove_df_a([true|Agenda]).
prove_df_a([(A,B)|Agenda]) :-
  !,
  findall(D,(clause(A,C),conj_append(C,B,D)),Children),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).
prove_df_a([A|Agenda]) :-
  findall(B,clause(A,B),Children),
  append(Children,Agenda,NewAgenda),
  prove_df_a(NewAgenda).
```

swapping arguments of append/3 turns this into a breadth-first meta-interpreter!

Proving as a search process:
bf agenda-based meta-interpreter

This time with answer substitution.

```
foo(X) :- bar(X).
```

```
?- findall(Body,clause(foo(Z),Body),Bodies).
Bodies = [bar(_G336)].
```

problem:
findall(Term,Goal,List) creates new variables in the instantiation of Term for the unbound variables in answers to Goal

trick:
store a(Literals,OriginalGoal) on agenda where OriginalGoal is a copy of the Goal

breadth-first

```
prove_bf(Goal):-
  prove_bf_a([a(Goal,Goal)],Goal).
prove_bf_a([a(true,Goal)|Agenda],Goal).
prove_bf_a([a((A,B),G)|Agenda],Goal):-!,
  findall(a(D,G),(clause(A,C),conj_append(C,B,D)),Children),
  append(Agenda,Children,NewAgenda),
prove_bf_a(NewAgenda,Goal).
prove_bf_a([a(A,G)|Agenda],Goal):-
  findall(a(B,G),clause(A,B),Children),
  append(Agenda,Children,NewAgenda),
  prove_bf_a(NewAgenda,Goal).
```

Goal will be instantiated with the correct answer substitutions

Proving as a search process:
forward vs backward chaining of if-then rules

backward chaining	forward chaining
from head to body	from body to head
search starts from where we want to be towards where we are	search starts from where we are to where we want to be
e.g. Prolog query answering	e.g. model construction

what's more efficient depends on structure of search space (cf. discussion on practical uses of var)

Proving as a search process:
forward chaining - bottom-up model construction

model of clauses defined by cl/1

```
model(M):- model([],M).
model(M0,M):-
  is_violated(Head,M0),!,
  disj_element(L,Head),
  model([L|M0],M).
model(M,M).

is_violated(H,M) :-
  cl((H:-B)),
  satisfied_body(B,M),
  not(satisfied_head(H,M)).
```

grounds literal from head

no more violated clauses (note the !)

grounds literal from body

add a literal from the head of a violated clause to the current model

a violated clause: body is true in the current model, but the head not

Proving as a search process:
forward chaining - auxiliaries

body is a conjunction of literals

```
satisfied_body(true,M).
satisfied_body(A,M) :-
  element(A,M).
satisfied_body((A,B),M) :-
  element(A,M),
  satisfied_body(B,M).
```

single disjunct

```
disj_element(X,X):-
  not(X=false),
  not(X=(One;TheOther)).
disj_element(X,(X;Ys)).
disj_element(X,(Y;Ys)):-
  disj_element(X,Ys).
```

false = empty disjunction

, and ; are right-associative operators: a;b;c=;(a,;(b,c))

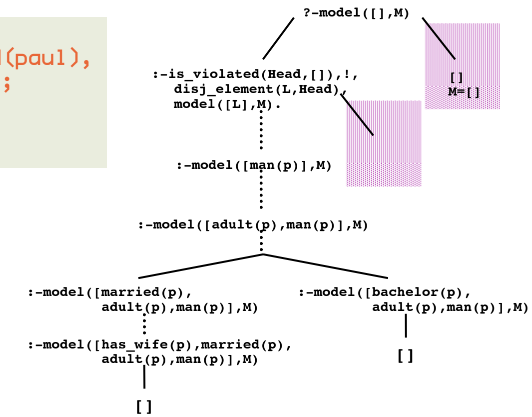
```
satisfied_head(A,M):-
  element(A,M).
satisfied_head((A;B),M) :-
  element(A,M).
satisfied_head((A;B),M) :-
  satisfied_head(B,M).
```


Proving as a search process:
forward chaining - example

```
cl((married(X);bachelor(X):-man(X),adult(X))).
cl((has_wife(X):-married(X),man(X))).
cl((man(paul):-true)).
cl((adult(paul):-true)).
```

```
?- model(M)
M = [has_wife(paul),married(paul),
     adult(paul),man(paul)];
M = [bachelor(paul),
     adult(paul),
     man(paul)]
```

two minimal models as there is a disjunction in the head



Proving as a search process:
forward chaining - range-restricted clauses

Our simple forward chainer cannot construct a model for following clauses:

```
cl((man(X);woman(X):-true)).
cl((false:-man(maria))).
cl((false:-woman(peter))).
```

an unground man(X) will be added to the model, which leads to the second clause being violated –which cannot be solved as it has an empty head

works only for clauses for which grounding the body also grounds the head

add literal to first clause, to enumerate possible values of X

```
cl((man(X);woman(X):-person(X))).
cl((person(maria):-true)).
cl((person(peter):-true)).
cl((false:-man(maria))).
cl((false:-woman(peter))).
```

range-restricted clause: all variables in head also occur in body

can be ensured by adding predicates that quantify over each variable's domain

```
?- model(M)
M = [man(peter),person(peter),woman(maria),person(maria)]
```


Proving as a search process:
forward chaining - subsets of infinite models

```
cl((append([],Y,Y):-list(Y))).
cl((append([X|Xs],Ys,[X|Zs]):-thing(X),append(Xs,Ys,Zs))).
cl((list([]):-true)).
cl((list([X|Y]):-thing(X),list(Y))).
cl((thing(a):-true)).
cl((thing(b):-true)).
cl((thing(c):-true)).
```

range-restricted version of append/3

```
model_d(D,M):-
  model_d(D,[],M).

model_d(0,M,M).
model_d(D,M0,M):-
  D>0,
  D1 is D-1,
  findall(H,is_violated(H,M0),Heads),
  satisfy_clauses(Heads,M0,M1),
  model_d(D1,M1,M).

satisfy_clauses([],M,M).
satisfy_clauses([H|Hs],M0,M):-
  disj_element(L,H),
  satisfy_clauses(Hs,[L|M0],M).
```

depth-bounded construction of submodel

Informed search:
best-first search

informed: use a heuristic estimate of the distance from a node to a goal given by predicate eval/2

```
search_best([Goal|RestAgenda],Goal):-
  goal(Goal).
search_best([CurrentNode|RestAgenda],Goal):-
  children(CurrentNode,Children),
  add_best(Children,RestAgenda,NewAgenda),
  search_best(NewAgenda,Goal).

add_best([],Agenda,Agenda).
add_best([Node|Nodes],Agenda,NewAgenda):-
 insert(Node,Agenda,TmpAgenda),
 add_best(Nodes,TmpAgenda,NewAgenda).
```

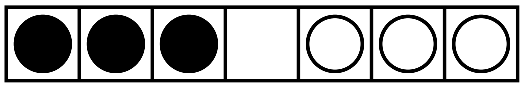
best-first: children of node are added according to heuristic (lowest value first)

Agenda is sorted

add_best(A,B,C): C contains the elements of A and B (B and C sorted according to eval/2)

```
insert(Node,Agenda,NewAgenda):-
  eval(Node,Value),
  insert(Value,Node,Agenda,NewAgenda).
insert(Value,Node,[],[Node]).
insert(Value,Node,[FirstNode|RestOfAgenda],[Node,FirstNode|RestOfAgenda]):-
  eval(FirstNode, FirstNodeValue),
  Value < FirstNodeValue.
insert(Value,Node,[FirstNode|RestOfAgenda],[FirstNode|NewRestOfAgenda]):-
  eval(FirstNode,FirstNodeValue),
  Value >= FirstNodeValue,
  insert(Value,Node,RestOfAgenda,NewRestOfAgenda).
```


Informed search:
best-first search on a puzzle



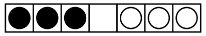
A tile may be moved to the empty spot if there are at most 2 tiles between it and the empty spot.

Find a series of moves that bring all the black tiles to the right of all the white tiles.

Cost of a move: 1 if no tiles were in between, otherwise amount of tiles jumped over.

Informed search:
best-first search on a puzzle - encoding

Board:



```
[b,b,b,e,w,w,w]
```

```
get_tile(Position,N,Tile) :-
  get_tile(Position,1,N,Tile).

get_tile([Tile|Tiles],N,N,Tile).
get_tile([Tile|Tiles],N0,N,FoundTile) :-
  N1 is N0+1,
  get_tile(Tiles, N1, N, FoundTile).
```

```
replace([Tile|Tiles],1,ReplacementTile, [ReplacementTile|Tiles]).
replace([Tile|Tiles],N,ReplacementTile, [Tile|RestOfTiles]):-
  N>1,
  N1 is N-1,
  replace(Tiles,N1,ReplacementTile,RestOfTiles).
```

Moves:

```
start_move(move(noparent, [b,b,b,e,w,w,w],0))
```

from / to / cost

Agenda items:

```
move_value(Move, Value)
```

heuristic evaluation of position reached by Move

Informed search:
best-first search on a puzzle - algorithm

```
tiles(ListOfPositions, TotalCost):-
  start_move(StartMove),
  eval(StartMove, Value),
  tiles([move_value(StartMove, Value)], FinalMove, [], VisitedMoves),
  order_moves(FinalMove, VisitedMoves, [], ListOfPositions,0, TotalCost).
```

acc for VisitedMoves

best-first search accumulating path

print path backwards from final move to start move

acc for ListOfPositions

acc for TotalCost

tiles(Agenda, LastMove, V0, V): goal can be reached from a move in Agenda where LastMove is the last move leading to the goal, and V is V0 + the set of moves tried.

```
tiles([move_value(LastMove,Value)|RestAgenda],LastMove,VisitedMoves,VisitedMoves):-
  goal(LastMove).
tiles([move_value(Move,Value)|RestAgenda],Goal,VisitedMoves, FinalVisitedMoves):-
  show_move(Move,Value),
  setof0(move_value(NextMove,NextValue),
         (next_move(Move,NextMove),eval(NextMove,NextValue)),
         Children),
  merge(Children,RestAgenda,NewAgenda),
  tiles(NewAgenda,Goal, [Move|VisitedMoves],FinalVisitedMoves).
```

finds sorted list of children with their evaluation

Informed search:
best-first search on a puzzle - encoding'

```
next_move(move(Position,LastPosition,LastCost),
          move(LastPosition,NewPosition,Cost)) :-
  get_tile(LastPosition, Ne, e),
  get_tile(LastPosition, Nbw, BW),
  not(BW=e),
  Diff is abs(Ne-Nbw),
  Diff<4,
  replace(LastPosition,Ne,BW,IntermediatePosition),
  replace(IntermediatePosition,Nbw,e,NewPosition),
  (Diff=1 -> Cost=1
   ; otherwise -> Cost is Diff-1
  ).
```

NewPosition is reached in one move from LastPosition with cost Cost

```
goal(Move):-
  eval(Move,0).

eval(move(OldPosition,Position,C),Value):-
  bLeftOfw(Position,Value).

bLeftOfw(Pos,Val):-
  findall((Nb,Nw),
          (get_tile(Pos,Nb,b),get_tile(Pos,Nw,w), Nb<Nw),L),
  length(L,Val).
```

sum of the number of black tiles to the left of each white tile

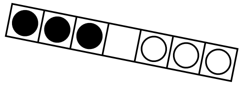
Informed search:
best-first search on a puzzle - auxiliaries

order_moves(FinalMove, VisitedMoves,Positions,FinalPositions,TotalCost,FinalTotalCost):
FinalPositions = Positions + connecting sequence of target positions from VisitedMoves ending in FinalMove's target position.
FinalTotalCost = TotalCost + total cost of moves added to Positions to obtain FinalPositions.

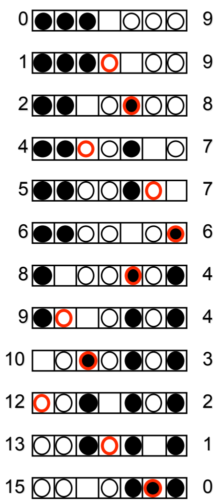
```
order_moves(move(noparent,StartPosition,0),
            VisitedMoves,Positions,
            [StartPositionPositions],TotalCost,TotalCost).

order_moves(move(FromPosition,ToPosition,Cost),
            VisitedMoves,Positions,
            FinalPositions,TotalCost,FinalTotalCost):-
  element(PreviousMove, VisitedMoves),
  PreviousMove = move(PreviousPosition, FromPosition,CostOfPreviousMove),
  NewTotalCost is TotalCost + Cost,
  order_moves(PreviousMove,VisitedMoves,
              [ToPosition|Positions],FinalPositions,NewTotalCost,FinalTotalCost).
```

Informed search:

best-first search on a puzzle - example run



```
?- tiles(M,C).
 [b,b,b,e,w,w,w]-9
 [b,b,b,w,e,w,w]-9
 [b,b,e,w,b,w,w]-8
 [b,b,w,w,b,e,w]-7
 [b,b,w,w,b,w,e]-7
 [b,b,w,w,e,w,b]-6
 [b,e,w,w,b,w,b]-4
 [b,w,e,w,b,w,b]-4
 [e,w,b,w,b,w,b]-3
 [w,w,b,e,b,w,b]-2
 [w,w,b,w,b,e,b]-1
M =[[b,b,b,e,w,w,w], [b,b,b,w,e,w,w],
    [b,b,e,w,b,w,w], [b,b,w,w,b,e,w],
    [b,b,w,w,b,w,e], [b,b,w,w,e,w,b],
    [b,e,w,w,b,w,b], [b,w,e,w,b,w,b],
    [e,w,b,w,b,w,b], [w,w,b,e,b,w,b],
    [w,w,b,w,b,e,b], [w,w,e,w,b,b,b]]
C = 15
```



Informed search:

optimal best search

Best-first search is not complete by itself: a heuristic might consistently assign lower values to the nodes on an infinite path

An A algorithm is a complete best-first search algorithm that aims at minimizing the total cost along a path from start to goal.

$$f(n) = g(n) + h(n)$$

actual cost so far: adds breadth-first flavor

estimate on further cost to reach goal: if optimistic (underestimating the cost), an optimal path will always be found. Such an algorithm is called A*.

$h(n)=0$: degenerates to breadth-first

Declarative Programming

5: natural language processing using DCGs

Definite clause grammars:

context-free grammars in Prolog

context-sensitive example:

```
noun,singular-->[turtle],singular.
singular,intransitive_verb-->[sleep]
```

one non-terminal on left-hand side

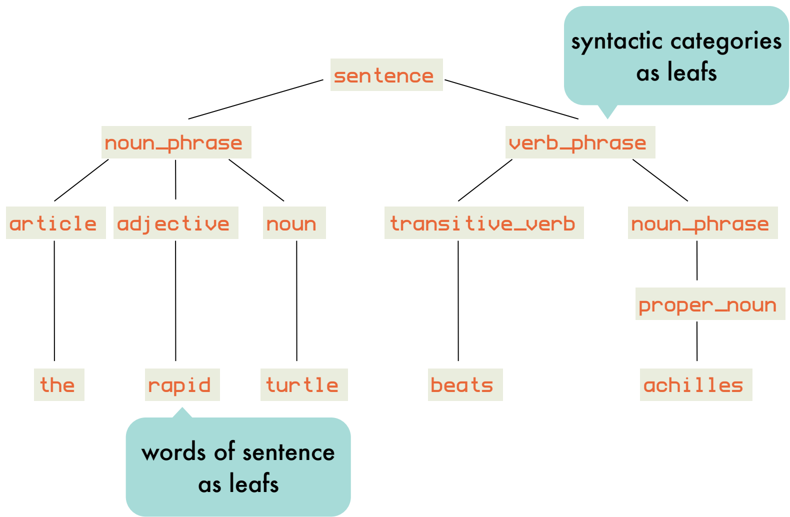
```
sentence   --> noun_phrase,verb_phrase.
noun_phrase    --> proper_noun.
noun_phrase    --> article,adjective,noun.
noun_phrase    --> article,noun.
verb_phrase    --> intransitive_verb.
verb_phrase    --> transitive_verb,noun_phrase.
article    --> [the].
adjective  --> [lazy].
adjective  --> [rapid].
proper_noun    --> [achilles].
noun    --> [turtle].
intransitive_verb--> [sleeps].
transitive_verb   --> [beats].
```

non-terminal defined by rule produces syntactic category

terminal: word in language

sentences generated by grammar are lists of terminals:
the lazy turtle sleeps, Achilles beats the turtle, the rapid turtle beats Achilles

Definite clause grammars:
parse trees for generated sentences



Definite clause grammars:
top-down construction of parse trees

similar to SLD-resolution!

start with NT and repeatedly replace NTS on right-hand side of an applicable rule until sentence is obtained as a list of terminals

DCG rules and Prolog clauses:
equivalence

sentence

```
[the, rapid, turtle, beats, achilles]
```

grammar rule

```
sentence --> noun_phrase,
             verb_phrase
```

```
verb-->[sleeps]
```

equivalent Prolog clause

```
sentence(S) :-
  noun_phrase(NP),
  verb_phrase(VP),
  append(NP,VP,S).
```

```
verb([sleeps]).
```

S is a sentence if some first part belongs to the noun_phrase category and some second part to the verb_phrase category

parsing

```
?- sentence([the,rapid,turtle,beats,achilles])
```


DCG rules and Prolog clauses:
built-in equivalence without append/3

grammar rule (meta-level)

```
sentence --> noun_phrase,
             verb_phrase
```

equivalent Prolog clause (object-level)

```
sentence(L,L0) :-
  noun_phrase(L,L1),
  verb_phrase(L1,L0).
```

L consists of a sentence followed by L0

parsing

```
?- phrase(sentence, L)
```

starting non-terminal

built-in meta-predicate calling sentence(L,[])

DCG rules and Prolog clauses:
summary and expressivity

	GRAMMAR	PARSING
META-LEVEL	s --> np,vp	?-phrase(s,L)
OBJECT-LEVEL	s(L,L0):- np(L,L1), vp(L1,L0)	?-s(L,[])

non-terminals can have arguments
goals can be put into the rules
no need for deterministic grammars
a single formalism for specifying syntax, semantics
parsing and generating

Expressivity of DCG rules:
non-terminals with arguments - plurality

```
sentence  --> noun_phrase(N),verb_phrase(N).
noun_phrase(N)  --> article(N),noun(N).
verb_phrase(N)  --> intransitive_verb(N).
article(singular)  --> [a].
article(singular)  --> [the].
article(plural) --> [the].
noun(singular)  --> [turtle].
noun(plural) --> [turtles].
intransitive_verb(singular)  --> [sleeps].
intransitive_verb(plural)--> [sleep].
```

arguments unify to express plurality agreement

```
phrase(sentence, [a,turtle,sleeps]). % yes
phrase(sentence, [the,turtles,sleep]). % yes
phrase(sentence, [the,turtles,sleeps]). % no
```


Expressivity of DCG rules:
non-terminals with arguments - parse trees

```
sentence(s(NP,VP))--> noun_phrase(NP),verb_phrase(VP).
noun_phrase(np(N))--> proper_noun(N).
noun_phrase(np(Art,Adj,N))   --> article(Art),adjective(Adj),
                                  noun(N).
noun_phrase(np(Art,N))   --> article(Art),noun(N).
verb_phrase(vp(IV))  --> intransitive_verb(IV).
verb_phrase(vp(TV,NP))   --> transitive_verb(TV),noun_phrase(NP).
article(art(the)) --> [the].
adjective(adj(lazy))  --> [lazy].
adjective(adj(rapid)) --> [rapid].
proper_noun(pn(achilles))--> [achilles].
noun(n(turtle))   --> [turtle].
intransitive_verb(iv(sleeps))--> [sleeps].
transitive_verb(tv(beats))   --> [beats].
```

```
?-phrase(sentence(T),[achilles,beats,the,lazy,turtle])

T = s(np(pn(achilles)),
      vp(tv(beats),
         np(art(the),
            adj(lazy),
            n(turtle))))
```


Expressivity of DCG rules:
goals in rule bodies

X_Y(N) if N is a number in [X..Y].

```
numeral(N) --> n1_999(N).
numeralN)  --> n1_9(N1),[thousand],n1_999(N2),{N is N1*1000+N2}.
n1_999(N)  --> n1_99(N).
n1_999(N)  --> n1_9(N1),[hundred],n1_99(N2),{N is N1*100+N2}.
n1_99(N)   --> n0_9(N).
n1_99(N)   --> n10_19(N).
n1_99(N)   --> n20_90(N).
n1_99(N)   --> n20_90(N1),n1_9(N2),{N is N1+N2}.
n0_9(0)--> [].
n0_9(N)--> n1_9(N).
n1_9(1)--> [one].
n1_9(2)--> [two].
  …
n10_19(10) --> [ten].
n10_19(11) --> [eleven].
  …
n20_90(20) --> [twenty].
n20_90(30) --> [thirty].
  …
```

regular goal enclosed by braces

```
n1_99(N,L,L0) :-
  n20_90(N1,L,L1),
  n1_9(N2,L1,L0),
  N is N1 + N2.
```

```
?-phrase(numeral(2211),N).
N =  [two,thousand,two,hundred,eleven]
```

Interpretation of natural language:

syntax and semantics

syntax

```
sentence --> determiner, noun, verb_phrase
sentence --> proper_noun, verb_phrase
verb_phrase --> [is], property
property --> [a], noun
property --> [mortal]
determiner --> [every]
proper_noun --> [socrates]
noun --> [human]
```

semantics

```
[every, human, is, mortal]
```

interpret a sentence: assign a clause to it

```
mortal(X):- human(X)
```

represents meaning of sentence

Interpretation of natural language:

interpreting sentences as clauses (I)

```
proper_noun(socrates) -->
                  [socrates]
```

the meaning of the proper noun 'Socrates' is the term socrates

```
property(X=>mortal(X)) --> [mortal].
```

operator X=>L: term X is mapped to literal L

the meaning of the property 'mortal' is a mapping from terms to literals containing the unary predicate mortal

```
verb_phrase(M) --> [is], property(M).
sentence([(L:-true)]) --> proper_noun(X),
                          verb_phrase(X=>L).
```

singleton clause list, cf. determiner 'some'

the meaning of a phrase (proper noun - verb) is a clause with empty body and of which the head is obtained by applying the meaning of the verb phrase to the meaning of the proper noun

```
?-phrase(sentence(C), [socrates,is,mortal]).
C = [(mortal(socrates):- true)]
```


Interpretation of natural language:

interpreting sentences as clauses (II)

```
sentence(C) --> determiner(M1,M2,C),
                noun(M1),
                verb_phrase(M2).
noun(X=>human(X)) --> [human].
```

```
determiner(X=>B, X=>H, [(H:- B)]) --> [every].
```

```
?-phrase(sentence(C), [every,human,is,mortal])
C = [(mortal(X):- human(X))]
```

the meaning of a determined sentence with determiner 'every' is a clause with the same variable in head and body

Interpretation of natural language:

interpreting sentences as clauses (III)

```
determiner(sk=>H1,sk=>H2,
           [(H1:-true),(H1:-true)] --> [some].
```

```
?-phrase(sentence(C), [some,humans,are,mortal])
C = [(human(sk):-true),(mortal(sk):-true)]
```

the meaning of a determined sentence with determiner 'some' are two clauses about the same individual (i.e., skolem constant)

Interpretation of natural language:
relational nature illustrated

```
?-phrase(sentence(C),S).
C = human(X):-human(X)
S = [every,human,is,a,human];
C = mortal(X):-human(X)
S = [every,human,is,mortal];
C = human(socrates):-true
S = [socrates,is,a,human];
C = mortal(socrates):-true
S = [socrates,is,mortal];
```

```
?-phrase(sentence(Cs),[D,human,is,mortal]).
D = every, Cs = [(mortal(X):-human(X))];
D = some, Cs = [(human(sk):-true),(mortal(sk):-true)]
```


Interpretation of natural language:
complete grammar with plurality agreement

```
:- op(600,xfy,'=>').
sentence(C) --> determiner(N,M1,M2,C), noun(N,M1),
verb_phrase(N,M2).
sentence([(L:- true)]) --> proper_noun(N,X),
verb_phrase(N,X=>L).
verb_phrase(s,M) --> [is], property(s,M).
verb_phrase(p,M) --> [are], property(p,M).
property(N,X=>mortal(X)) --> [mortal].
property(s,M) --> noun(s,M).
property(p,M) --> noun(p,M).
determiner(s, X=>B , X=>H, [(H:- B)]) --> [every].
determiner(p, sk=>H1, sk=>H2, [(H1 :- true),(H2 :- true)]) -->[some].
proper_noun(s,socrates) --> [socrates].
noun(s,X=>human(X)) --> [human].
noun(p,X=>human(X)) --> [humans].
noun(s,X=>living_being(X)) --> [living],[being].
noun(p,X=>living_being(X)) --> [living],[beings].
```


Interpretation of natural language:
shell for building up and querying rule base

grammar for queries

```
question(Q) --> [who], [is], property(s,X=>Q)
question(Q) --> [is], proper_noun(N,X), property(N,X=>Q)
question((Q1,Q2)) --> [are], [some], noun(p,sk=>Q1),
                      property(p,sk=>Q2)
```

shell

```
nl_shell(RB) :- get_input(Input), handle_input(Input,RB).

handle_input(stop,RB) :- !.
handle_input(show,RB) :- !, show_rules(RB), nl_shell(RB).
handle_input(Sentence,RB) :- phrase(sentence(Rule),Sentence),
                             nl_shell([Rule|RB]).
handle_input(Question,RB) :- phrase(question(Query),Question),
                             prove_rb(Query,RB),!
                             transform(Query,Clauses),
                             phrase(sentence(Clauses),Answer),
                             show_answer(Answer),
                             nl_shell(RB).
handle_input(Error,RB) :-    show_answer('no'), nl_shell(RB).
```

add new rule

question that can be solved

transform instantiated query (conjuncted literals) to list of clauses with empty body

generate nl

Interpretation of natural language:
shell for building up and querying rule base - aux

convert rule to natural language sentence

```
show_rules([]).
show_rules([R|Rs]) :-
   phrase(sentence(R),Sentence),
   show_answer(Sentence),
   show_rules(Rs).
get_input(Input) :-
   write('? '),read(Input).
   show_answer(Answer) :-
   write('! '),write(Answer), nl.
```

```
show_answer(Answer) :- write('!'),nl.
```

```
get_input(Input) :- write('?'),read(Input).
```

convert query to list of clauses for which natural language sentences can be generated

```
transform((A,B),[(A:-true)|Rest]):-!,
  transform(B,Rest).
transform(A,[(A:-true)]).
```

Interpretation of natural language:

shell for building up and querying rule base - interpreter

```
prove(true,RB) :- !.
prove((A,B),RB) :- !,
  prove(A,RB),prove(B,RB).
prove(A,RB) :-
  find_clause((A:-B),RB),
  prove(B,RB).

find_clause(C, [R|Rs]) :-
  copy_element(C,R).
find_clause(C, [R|Rs]) :-
  find_clause(C,Rs).

copy_element(X,Ys) :- element(X1,Ys),
                      copy_term(X1,X).
```

handy when storing rule base in list

finds a clause in the rule base, but without instantiating its variables (rule can be used multiple times, rules can share variables)

copy_term(*+In*, *-Out*)
Create a version if *In* with renamed (fresh) variables and unify it to *Out*.

Interpretation of natural language:

shell for building up and querying rule base - example

```
? [every,human,is,mortal]
? [socrates,is,a,human]
? [who,is,mortal]
! [socrates,is,mortal]
? [some,living,beings,are,humans]
? [are,some,living,beings,mortal]
! [some,living,beings,are,mortal]
```

built-in repeat/1 succeeds indefinitely

possible improvement: apply idiom of failure-driven loop to avoid memory issues

```
shell :- repeat, get_input(X), handle_input(X).
handle_input(stop) :- !.
handle_input(X) :- /* handle */, fail.
```

causes backtracking to repeat literal

Declarative Programming

6: reasoning with incomplete information: default reasoning, abduction

Reasoning with incomplete information:

overview

reasoning that leads to conclusions that are plausible, but not guaranteed to be true because not all information is available

Such reasoning is unsound. Deduction is sound, but only makes implicit information explicit.

default reasoning	abduction	induction
assume normal state of affairs, unless there is evidence to the contrary	choose between several explanations that explain an observation	generalize a rule from a number of similar observations
"If something is a bird, it flies."	*"I flipped the switch, but the light doesn't turn on. The bulb mist be broken"*	*"The sky is full of dark clouds. It will rain."*

Default reasoning:

Tweety is a bird. Normally, birds fly. Therefore, Tweety flies.

```
bird(tweety).
flies(X) :- bird(X), normal(X).
```

has three models:

```
{bird(tweety)}
{bird(tweety), flies(tweety)}
{bird(tweety), flies(tweety), normal(tweety)}
```

bird(tweety) is the only logical conclusion of the program because it occurs in every model.

If we want to conclude flies(tweety) through deduction, we have to state normal(tweety) explicitly. Default reasoning assumes something is normal, unless it is known to be abnormal.

Default reasoning:

A more natural formulation using abnormal/1

```
bird(tweety).
flies(X) ; abnormal(X) :- bird(X).
```

indefinite clause

has two minimal models:

```
{bird(tweety), flies(tweety)}
{bird(tweety), abnormal(tweety)}
```

model 2 is model of the general clause:

```
abnormal(X) :- bird(X), not(flies(X)).
```

model 1 is model of the general clause:

```
flies(X):-bird(X), not(abnormal(X)).
```

using negation as failure: tweety flies if it cannot be proven that he is abnormal

```
bird(tweety).
flies(X):-bird(X), not(abnormal(X)).
ostrich(tweety).
abnormal(X) :- ostrich(X).
```

tweety no longer flies, he is an ostrich: the default rule (birds fly) is cancelled by the more specific rule (ostriches)

Default reasoning:

non-monotonic form of reasoning

new information can invalidate previous conclusions:

```
bird(tweety).
flies(X):-bird(X),not(abnormal(X)).
```

```
bird(tweety).
flies(X):-bird(X),not(abnormal(X)).
ostrich(tweety).
abnormal(X) :- ostrich(X).
```

Not the case for deductive reasoning, which is monotonic in the following sense:

$$Th \vdash p \Rightarrow Th \cup \{q\} \vdash p$$

$$Closure(Th) = \{p \mid Th \vdash p\}$$

$$Th1 \subseteq Th2 \Rightarrow Closure(Th1) \subseteq Closure(Th2)$$

Default reasoning:

without not/1, using a meta-interpreter

problematic: e.g., floundering but also because it has no clear declarative semantics

Distinguish regular rules (without exceptions) from default rules (with exceptions.)

Only apply a default rule when it does not lead to an inconsistency.

```
default((flies(X) :- bird(X))).
rule((not(flies(X)) :- penguin(X))).
rule((bird(X) :- penguin(X))).
rule((penguin(tweety) :- true)).
rule((bird(opus) :- true)).
```

Default reasoning:
using a meta-interpreter

```
explain(F,E):-
  explain(F, [],E).
explain(true,E,E) :- !.
explain((A,B),E0,E) :- !,
  explain(A,E0,E1),
  explain(B,E1,E).
explain(A,E0,E):-
  prove(A,E0,E).
explain(A,E0, [default((A:-B))|E]):-
  default((A:-B)),
  explain(B,E0,E),
  not(contradiction(A,E)).
```

E explains F: lists the rules used to prove F

prove using regular rules

prove using default rules

do not use a default to prove A (or not(A)) if you can prove not(A) (or A) using regular rules

```
prove(true,E,E) :- !.
prove((A,B),E0,E) :- !,
  prove(A,E0,E1),
  prove(B,E1,E).
prove(A,E0, [rule((A:-B))|E]):-
  rule((A:-B)),
  prove(B,E0,E).
```

```
contradiction(not(A),E) :- !,
  prove(A,E,_).
contradiction(A,E):-
  prove(not(A),E,_).
```


Default reasoning:
using a meta-interpreter, Opus example

```
default((flies(X) :- bird(X))).
   rule((not(flies(X)) :- penguin(X))).
   rule((bird(X) :- penguin(X))).
   rule((penguin(tweety) :- true)).
   rule((bird(opus) :- true)).
```

```
?- explain(flies(X),E)
X=opus
E=[default((flies(opus) :- bird(opus))),
      rule((bird(opus) :- true))]

?- explain(not(flies(X)),E)
X=tweety
E=[rule((not(flies(tweety)) :- penguin(tweety))),
   rule((penguin(tweety) :- true))]
```

default rule has been cancelled

Default reasoning:
using a meta-interpreter, Dracula example

```
default((not(flies(X)) :- mammal(X))).
default((flies(X) :- bat(X))).
default((not(flies(X)) :- dead(X))).
   rule((mammal(X) :- bat(X))).
   rule((bat(dracula) :- true)).
   rule((dead(dracula) :- true)).
```

```
?-explain(flies(dracula),E)
E=[default((flies(dracula) :- bat(dracula))),
      rule((bat(dracula) :- true))]
```

dracula flies because bats typically fly

```
?-explain(not(flies(dracula)),E)
E=[default((not(flies(dracula)) :- mammal(dracula)))
      rule((mammal(dracula) :- bat(dracula))),
      rule((bat(dracula) :- true))]
E=[default((not(flies(dracula)) :- dead(dracula)))
      rule((dead(dracula) :- true))]
```

dracula doesn't fly because mammals typically don't

dracula doesn't fly because dead things typically don't

Default reasoning:
using a revised meta-interpreter

need a way to cancel particular defaults in certain situations: bats are flying mammals although the default is that mammals do not fly

name associated with default rule

```
default(mammals_dont_fly(X), (not(flies(X)):-mammal(X))).
default(bats_fly(X), (flies(X):-bat(X))).
default(dead_things_dont_fly(X), (not(flies(X)):-dead(X))).
   rule((mammal(X):-bat(X))).
   rule((bat(dracula):-true)).
   rule((dead(dracula):-true)).
   rule((not(mammals_dont_fly(X)):-bat(X))).
   rule((not(bats_fly(X)):-dead(X))).
```

Default reasoning:
using a revised meta-interpreter

need a way to cancel particular defaults in certain situations: bats are flying mammals although the default is that mammals do not fly

name associated with default rule

```
default(mammals_dont_fly(X), (not(flies(X)):-mammal(X))).
default(bats_fly(X), (flies(X):-bat(X))).
default(dead_things_dont_fly(X), (not(flies(X)):-dead(X))).
   rule((mammal(X):-bat(X))).
   rule((bat(dracula):-true)).
   rule((dead(dracula):-true)).
   rule((not(mammals_dont_fly(X)):-bat(X))).
   rule((not(bats_fly(X)):-dead(X))).
```

rule cancels the mammals_dont_fly default

Default reasoning:
using a revised meta-interpreter

explanations keep track of names rather than default rules

```
explain(A,E0,[default(Name)|E]):-
  default(Name,(A:- B)),
  explain(B,E0,E),
  not(contradiction(Name,E)),
  not(contradiction(A,E)).
```

default rule is not cancelled in this situation: e.g., do not use default named bats_fly(X) if you can prove not(bats_fly(X))

dracula can not fly after all

```
?-explain(flies(dracula),E)
no
?-explain(not(flies(dracula)),E)
E=[default(dead_things_dont_fly(dracula)),
      rule((dead(dracula):- true))]
```


Default reasoning:
Dracula revisited

using meta-interpreter

```
default(mammals_dont_fly(X), (not(flies(X)):-mammal(X))).
default(bats_fly(X), (flies(X):-bat(X))).
default(dead_things_dont_fly(X), (not(flies(X)):-dead(X))).
   rule((mammal(X):-bat(X))).
   rule((bat(dracula):-true)).
   rule((dead(dracula):-true)).
   rule((not(mammals_dont_fly(X)):-bat(X))).
   rule((not(bats_fly(X)):-dead(X))).
```

typical case is a clause that is only applicable when it does not lead to inconsistencies; applicability can be restricted using clause names

using naf

```
notflies(X):-mammal(X),not(flying_mammal(X)).
flies(X):-bat(X),not(nonflying_bat(X)).
notflies(X):-dead(X),not(flying_deadthing(X)).
mammal(X):-bat(X).
bat(dracula).
dead(dracula).
flying_mammal(X):-bat(X).
nonflying_bat(X):-dead(X).
```

typical case is general clause that negates abnormality predicate

Abduction:
given a theory T and an observation O, find an explanation E such that T∪E⊨O

T
```
likes(peter,S) :- student_of(S,peter).
likes(X,Y) :- friend(X,Y).
```

O
```
likes(peter,paul)
```

E1 `{student_of(paul,peter)}`

E2 `{friend(peter,paul)}`

Default reasoning makes assumptions about what is false (e.g., tweety is not an abnormal bird), abduction can also make assumptions about what is true.

```
{(likes(X,Y) :- friendly(Y)),
 friendly(paul)}
```

another possibility, but abductive explanations are usually restricted to ground literals with predicates that are undefined in the theory (abducibles)

Abduction:
abductive meta-interpreter

Theory ∪ Explanation ⊨ Observation

Try to prove Observation from theory, when a literal is encountered that cannot be resolved (an abducible), add it to the Explanation.

```
abduce(O,E):-
  abduce(O,[],E).
abduce(true,E,E) :- !.
abduce((A,B),E0,E) :- !,
  abduce(A,E0,E1),
  abduce(B,E1,E).
abduce(A,E0,E):-
  clause(A,B),
  abduce(B,E0,E).
abduce(A,E,E):-
  element(A,E).
abduce(A,E, [A|E]):-
  not(element(A,E)),
  abducible(A).
abducible(A):-
  not(clause(A,B)).
```

A already assumed

A can be assumed if it was not already assumed and it is an abducible.

```
likes(peter,S) :- student_of(S,peter).
likes(X,Y) :- friend(X,Y).

?-abduce(likes(peter,paul),E)
E = [student_of(paul,peter)];
E = [friend(paul,peter)]
```


Abduction:
abductive meta-interpreter and negation

general clauses

```
flies(X) :- bird(X), not(abnormal(X)).
abnormal(X) :- penguin(X).
bird(X) :- penguin(X).
bird(X) :- sparrow(X).

?-abduce(flies(tweety),E)
E = [not(abnormal(tweety)),penguin(tweety)];
E = [not(abnormal(tweety)),sparrow(tweety)];
```

inconsistent with theory as penguins are abnormal

abnormal/1 not an abducible

Since no clause is found for not(abnormal(tweety)), it is added to the explanation.

Abduction:
first attempt at abduction with negation

extend abduce/3 with negation as failure:

```
abduce(not(A),E,E):-
  not(abduce(A,E,E)).
```

do not add negated literals to the explanation:

```
abducible(A):-
  A \= not(X),
  not(clause(A,B)).
```

```
flies(X) :- bird(X), not(abnormal(X)).
abnormal(X) :- penguin(X).
bird(X) :- penguin(X).
bird(X) :- sparrow(X).

?-abduce(flies(tweety),E)
E = [sparrow(tweety)]
```


Abduction:
first attempt at abduction with negation: FAILED

any explanation of bird(tweety) will also be an explanation of flies1(tweety):

```
flies1(X):- not(abnormal(X)),bird(X)
abnormal(X) :- penguin(X).
bird(X) :- penguin(X).
bird(X) :- sparrow(X).
```

reversed order of literals

the fact that abnormal(tweety) is to be considered false, is not reflected in the explanation:

```
?- abduce(not(abnormal(tweety)),[],[])
true .
```

```
abduce(not(A),E,E):-
  not(abduce(A,E,E)).
```

assumes the explanation is already complete

Abduction:
final abductive meta-interpreter: abduce/3

```
abduce(true,E,E) :- !.
abduce((A,B),E0,E) :- !,
  abduce(A,E0,E1),
  abduce(B,E1,E).
abduce(A,E0,E):-
  clause(A,B),
  abduce(B,E0,E).
abduce(A,E,E):-
  element(A,E).
abduce(A,E, [A|E]):-
  not(element(A,E)),
  abducible(A),
  not(abduce_not(A,E,E)).
abduce(not(A),E0,E):-
  not(element(A,E0)),
  abduce_not(A,E0,E).
```

```
abducible(A):-
  A \= not(X),
  not(clause(A,B)).
```

A already assumed

A can be assumed if it was not already, it is abducible, E doesn't explain not(A)

only assume not(A) if A was not already assumed, ensure not(A) is reflected in the explanation

Abduction:
final abductive meta-interpreter: abduce_not/3

```
abduce_not((A,B),E0,E):-
  !,
  abduce_not(A,E0,E) ;
  abduce_not(B,E0,E).
abduce_not(A,E0,E):-
  setof(B,clause(A,B),L),
  abduce_not_list(L,E0,E).
abduce_not(A,E,E):-
  element(not(A),E).
abduce_not(A,E, [not(A)|E]):-
  not(element(not(A),E)),
  abducible(A),
  not(abduce(A,E,E)).
abduce_not(not(A),E0,E):-
  not(element(not(A),E0)),
  abduce(A,E0,E).
```

```
abduce_not_list([],E,E).
abduce_not_list([B|Bs],E0,E):-
  abduce_not(B,E0,E1),
  abduce_not_list(Bs,E1,E).
```

disjunction: a negation conjunction can be explained by explaining A or by explaining B

not(A) is explained by explaining not(B) for **every** A:-B

not(A) already assumed

assume not(A) if not already so, A is abducible and E does not already explain A

explain not(not(A)) by explaining A

Abduction:
final abductive meta-interpreter: example

```
flies(X) :- bird(X),not(abnormal(X)).
flies1(X) :- not(abnormal(X)),bird(X).
abnormal(X) :- penguin(X).
abnormal(X) :- dead(X).
bird(X) :- penguin(X).
bird(X) :- sparrow(X).
```

```
?- abduce(flies(tweety),E).
E = [not(penguin(tweety)),
     not(dead(tweety)),
     sparrow(tweety)]

?- abduce(flies1(tweety),E).
E = [sparrow(tweety),
     not(penguin(tweety)),
     not(dead(tweety))]
```

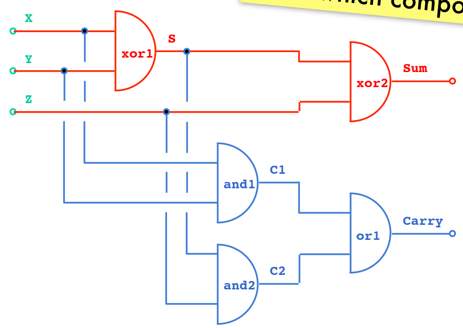
now abduces as expected

Abduction:
diagnostic reasoning

Theory: system description
Observation: input values, output values
Explanation: diagnosis=hypothesis about which components are faulty

3-bit adder

usually what has to be carried on from previous computation



Theory describing normal operation

```
adder(X,Y,Z,Sum,Carry) :-
  xor(X,Y,S),
  xor(Z,S,Sum),
  and(X,Y,C1),and(Z,S,C2),
  or(C1,C2,Carry).
```

```
xor(0,0,0).    and(0,0,0).    or(0,0,0).
xor(0,1,1).    and(0,1,0).    or(0,1,1).
xor(1,0,1).    and(1,0,0).    or(1,0,1).
xor(1,1,0).    and(1,1,1).    or(1,1,1).
```

Abduction:
diagnostic reasoning - fault model

describes how each component can behave in a faulty manner

```
fault(NameComponent=State)
```

```
adder(N,X,Y,Z,Sum,Carry):-
  xorg(N-xor1,X,Y,S),
  xorg(N-xor2,Z,S,Sum),
  andg(N-and1,X,Y,C1),
  andg(N-and2,X,S,C2),
  org(N-or1,C1,C2,Carry).
```

can be nested: subSystemName-componentName

correct behavior

```
xorg(N,X,Y,Z) :- xor(X,Y,Z).
xorg(N,0,0,1) :- fault(N=s1).
xorg(N,0,1,0) :- fault(N=s0).
xorg(N,1,0,0) :- fault(N=s0).
xorg(N,1,1,1) :- fault(N=s1).

xandg(N,X,Y,Z):- and(X,Y,Z).
xandg(N,0,0,1):- fault(N=s1).
xandg(N,0,1,1) :- fault(N=s1).
xandg(N,1,0,1):- fault(N=s1).
xandg(N,1,1,0) :- fault(N=s0)

org(N,X,Y,Z):- or(X,Y,Z).
org(N,0,0,1):- fault(N=s1).
org(N,0,1,0) :- fault(N=s0).
org(N,1,0,0):- fault(N=s0).
org(N,1,1,0) :- fault(N=s0).
```

faulty behavior

s0: output stuck at 0,
s1: output stuck at 1

Abduction:
diagnostic reasoning - diagnoses for faulty adder

```
diagnosis(Observation,Diagnosis):-
  abduce(Observation,Diagnosis).
```

adder(N,X,Y,Z,Sum,Carry): both Sum and Carry are wrong

obvious diagnosis: outputs of adder are stuck

```
?-diagnosis(adder(a,0,0,1,0,1),D).
D = [fault(a-or1=s1),fault(a-xor2=s0)];
D = [fault(a-and2=s1),fault(a-xor2=s0)];
D = [fault(a-and1=s1),fault(a-xor2=s0)];
D = [fault(a-and2=s1),fault(a-and1=s1),fault(a-xor2=s0)];
D = [fault(a-or1=s1),fault(a-and2=s0), fault(a-xor1=s1)];
D = [fault(a-and1=s1),fault(a-xor1=s1)];
D = [fault(a-and2=s0),fault(a-and1=s1), fault(a-xor1=s1)];
D = [fault(a-xor1=s1)]
```

most plausible as only one faulty component accounts for entire fault

Declarative semantics for incomplete information:
completing incomplete programs

semantics and proof theory for the not in a general clause will be discussed ~~later~~ NOW

problem

can no longer express

```
married(X); bachelor(X) :- man(X), adult(X).
man(john). adult(john).
```

characteristic of indefinite clauses

which had two minimal models

```
{man(john),adult(john),married(john)}
{man(john),adult(john),bachelor(john)}
{man(john),adult(john),married(john),bachelor(john)}
```

general clauses

definite clause containing not

first model is minimal model of **general** clause

```
married(X) :- man(X), adult(X), not bachelor(X).
```

second model is minimal model of **general** clause

```
bachelor(X) :- man(X), adult(X), not married(X).
```

to prove that someone is a bachelor, prove that he is a man and an adult, and prove that he is not a bachelor

Declarative semantics for incomplete information:
completing incomplete programs

A program P is "complete" if for every (ground) fact f, either P ⊨ f or P ⊨ ¬f

unique minimal model

Transform an incomplete program into a complete one, that captures the intended meaning of the original program.

possible transformations

closed world assumption	predicate completion
straightforward	ok for general clauses (with negation in body)
ok for definite clauses (without negation)	may lead to inconsistencies if the program is not stratified

Completing incomplete programs: *closed world assumption*

everything that is not known to be true, must be false

motivation: in general, there are more false statements that can be made than true statements

do not say something is not true, simply say nothing about it

Completing incomplete programs: *closed world assumption*

everything that is not known to be true, must be false

$CWA(P) = P \cup \{:\!-A \mid A \in B_P \wedge P \not\models A\}$

the clause "false :-A" is only true under interpretations in which A is false

CWA-complement of a program P (i.e, CWA(P)-P): explicitly assume that every ground atom A that does not follow from P is false

Completing incomplete programs: *closed world assumption - example*

P
```
likes(peter,S) :- student_of(S,peter).
student_of(paul,peter).
```

B_P
```
{likes(peter,peter),likes(peter,paul),
 likes(paul,peter),likes(paul,paul),
 student_of(peter,peter),student_of(peter,paul),
 student_of(paul,peter),student_of(paul,paul)}
```

only the black atoms are relevant for determining whether an interpretation is a model of every ground instance of every clause

models
```
{student_of(paul,peter),likes(peter,paul)}
{student_of(paul,peter),likes(peter,paul),likes(peter,peter)}
{student_of(paul,peter),likes(peter,paul),
 student_of(peter,peter),likes(peter,peter)}
...
```

there are still 4 orange atoms remaining which can each be added (or not) freely to the above interpretations

in total: 3*2^4=48 models for such a simple program!

$P \models A$
```
likes(peter,paul)
student_of(paul,peter)
```


Completing incomplete programs: *closed world assumption - example*

P
```
likes(peter,S) :- student_of(S,peter).
student_of(paul,peter).
```

B_P
```
{likes(peter,peter),likes(peter,paul),
 likes(paul,peter),likes(paul,paul),
 student_of(peter,peter),student_of(peter,paul),
 student_of(paul,peter),student_of(paul,paul)}
```

$P \models A$
```
likes(peter,paul)
student_of(paul,peter)
```

CWA(P)
```
likes(peter,S) :- student_of(S,peter).
student_of(paul,peter).
:- student(paul,paul).
:- student(peter,paul).
:- student(peter,peter).
:- likes(paul,paul).
:- likes(paul,peter).
:- likes(peter,peter).
```

is a complete program: every ground atom from B_P is assigned true or false

has only 1 model: {student_of(paul,peter),likes(peter,paul)} which is declared the intended model of the program (also obtained as the intersection of all models)

Completing incomplete programs: *closed world assumption - inconsistency*

when applied to indefinite and general clauses

P
```
bird(tweety).
flies(X);abnormal(X) :- bird(X).
```

B_P
```
{bird(tweety),abnormal(tweety),flies(tweety)}
```

models
```
{bird(tweety),flies(tweety)}
{bird(tweety),abnormal(tweety)}
{bird(tweety),abnormal(tweety),flies(tweety)}
```

$P \models A$
```
bird(tweety)
```

CWA(P)
```
bird(tweety).
flies(X);abnormal(X) :- bird(X).
:-abnormal(tweety).
:-flies(tweety)
```

CWA(P) is inconsistent

no longer has a model because, in order for the second clause to be true under an interpretation, its head needs to be true given that its body is already true due to the first clause

Completing incomplete programs: *predicate completion - idea*

turn implications (if) into equivalences (iff) by completing clauses (with their and-only-if part)

regard each clause as part of the complete definition of a predicate

only clause defining likes/2:

P
```
likes(peter,S) :- student(S,peter).
```

its completion:

$\forall X \forall S\ likes(X,S) \leftrightarrow X = peter \land student(S,peter)$

in clausal form:

Comp(P)
```
likes(peter,S) :- student(S,peter).
X=peter :- likes(X,S).
student(S,peter) :- likes(X,S)
```


Completing incomplete programs: *predicate completion - algorithm*

```
likes(peter,S) :- student_of(S,peter).
student_of(paul,peter).
```

1 ensure each argument of each clause head is a distinct variable

add literals Var=Term to body

```
likes(X,S) :- X=peter,student_of(S,peter).
student_of(X,Y) :- X=paul,Y=peter
```

2 if there are several clauses for a predicate, combine them into a single formula

use disjunction in implication's body if there are multiple clauses for a predicate

$\forall X \forall Y\ likes(X,Y) \leftarrow X=peter \land student\_of(Y,peter))$

$\forall X \forall Y\ student\_of(X,Y) \leftarrow X=paul \land Y=peter$

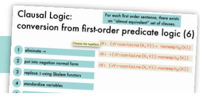
3 turn the implication into an equivalence

if a predicate without definition is used in a body (e.g. p/1), add $\forall X\ \neg p(X)$

$\forall X \forall Y\ likes(X,Y) \leftrightarrow X=peter \land student\_of(Y,peter))$

$\forall X \forall Y\ student\_of(X,Y) \leftrightarrow X=paul \land Y=peter$

4 convert to clausal form



Completing incomplete programs: *predicate completion - algorithm*

```
likes(peter,S) :- student_of(S,peter).
student_of(paul,peter).
```

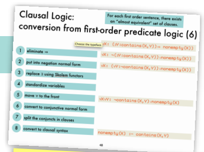
3 turn the implication into an equivalence

if a predicate without definition is used in a body (e.g. p/1), add $\forall X\ \neg p(X)$

$\forall X \forall Y\ likes(X,Y) \leftrightarrow X=peter \land student\_of(Y,peter))$

$\forall X \forall Y\ student\_of(X,Y) \leftrightarrow X=paul \land Y=peter$

4 convert to clausal form




```
likes(peter,S):-student_of(S,peter).
X=peter:-likes(X,S).
student_of(S,peter):-likes(X,S).
student_of(paul,peter).
X=paul:-student_of(X,Y).
Y=peter:-student_of(X,Y).
```

has the single model {student_of(paul,peter), likes(peter,paul)}

for definite clauses, CWA(P) and Comp(P) have same model

Completing incomplete programs: *predicate completion - existential variables*

if a predicate without definition is used in a body (e.g. p/1), add $\forall X\ \neg p(X)$

3 turn the implication into an equivalence

careful with variables in a body that do not occur in the head

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

$\forall X \forall Y\ ancestor(X,Y) \leftrightarrow (parent(X,Y) \vee$

$(\exists Z\ parent(X,Z) \wedge ancestor(Z,Y))))$

use second form because all clauses must have the same head

$\forall X \forall Y \forall Z\ ancestor(X,Y) \leftarrow parent(X,Z) \wedge ancestor(Z,Y)$

$\forall X \forall Y\ ancestor(X,Y) \leftarrow \exists Z\ parent(X,Z) \wedge ancestor(Z,Y))$

$\forall Z{:}q \leftarrow p(Z)$
$\forall Z{:}q \vee \neg p(Z)$
$q \vee \forall Z{:}\neg p(Z)$
$q \vee \exists Z{:}p(Z)$

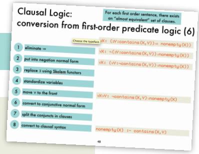
Completing incomplete programs: *predicate completion - existential variables*

3 turn the implication into an equivalence

$\forall X \forall Y\ ancestor(X,Y) \leftrightarrow (parent(X,Y) \vee$

$(\exists Z\ parent(X,Z) \wedge ancestor(Z,Y))))$

4 convert to clausal form




```
ancestor(X,Y):-parent(X,Y).
ancestor(X,Y):-parent(X,Z),ancestor(Z,Y).
parent(X,Y);parent(X,pa(X,Y)):-ancestor(X,Y).
parent(X,Y);ancestor(pa(X,Y),Y):-ancestor(X,Y).
```

Skolem functor
$\forall X \exists Y : loves(X,Y)$
$\forall X{:}loves(X,person\_loved\_by(X))$

Completing incomplete programs: *predicate completion - negation*

```
bird(tweety).
flies(X):-bird(X),not(abnormal(X)).
```

1 ensure each argument of each clause head is a distinct variable

```
bird(X):-X=tweety.
flies(X):-bird(X),not(abnormal(X)).
```

2 if there are several clauses for a predicate, combine them into a single formula

$\forall X\ bird(X) \leftarrow X{=}tweety.$
$\forall X\ flies(X) \leftarrow bird(X) \wedge \neg abnormal(X)$

3 turn the implication into an equivalence

if a predicate without definition is used in a body (e.g. p/1), add $\forall X\ \neg p(X)$

$\forall X\ bird(X) \leftrightarrow X{=}tweety.$

$\forall X\ flies(X) \leftrightarrow bird(X) \wedge \neg abnormal(X).$

$\forall X\ \neg abnormal(X)$

Completing incomplete programs: *predicate completion - negation*

```
bird(tweety).
flies(X):-bird(X),not(abnormal(X)).
```

if a predicate without definition is used in a body (e.g. p/1), add $\forall X\ \neg p(X)$

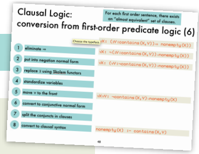
3 turn the implication into an equivalence

$\forall X\ bird(X) \leftrightarrow X{=}tweety.$

$\forall X\ flies(X) \leftrightarrow bird(X) \wedge \neg abnormal(X).$

$\forall X\ \neg abnormal(X)$

4 convert to clausal form




```
bird(tweety).
X=tweety:-bird(X).
flies(X);abnormal(X):-bird(X).
bird(X):-flies(X).
:-flies(X),abnormal(X).
:-abnormal(X).
```

has the single model {bird(tweety),flies(tweety)}

Completing incomplete programs:
predicate completion - inconsistency

Comp(P) is inconsistent for certain **unstratified** P

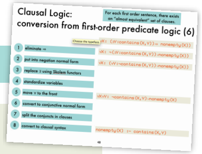
```
wise(X):-not(teacher(X)).
teacher(peter):-wise(peter).
```

if a predicate without definition is used in a body (e.g. p/1), add $\forall X\ \neg p(X)$

3 turn the implication into an equivalence

$\forall X\ wise(X) \leftrightarrow \neg teacher(X)$

$\forall X\ teacher(X) \leftrightarrow X = peter \wedge wise(peter)$



4 convert to clausal form

```
wise(X);teacher(X).
:-wise(X),teacher(X).
teacher(peter):-wise(peter).
X=peter:-teacher(X).
wise(peter):-teacher(X).
```

inconsistent!

Completing incomplete programs:
stratified programs

if P is stratified then Comp(P) is consistent

sufficient but not necessary: there are non-stratified P's for which Comp(P) is consistent

organize the program in layers (strata);
do not allow the programmer to negate a predicate that is not yet completely defined (in a lower stratum)

A program P is stratified if its predicate symbols can be partitioned into disjoint sets $S_0, \ldots, S_n$
such that for each clause $p(\ldots) \leftarrow L_1,\ldots,L_j$ where $p \in S_k$, any literal L_j is such that
if $L_j = q(\ldots)$ then $q \in S_0 \cup \ldots \cup S_k$
if $L_j = \neg q(\ldots)$ then $q \in S_0 \cup \ldots \cup S_{k-1}$

Completing incomplete programs:
soundness result for SLDNF-resolution

$P \vdash_{SLDNF} q \Rightarrow Comp(P) \models q$

completeness result only holds for a subclass of programs

Declarative Programming

7: inductive reasoning

Inductive reasoning: *overview*

infer general rules from specific observations

Given

B: background theory (clauses of logic program)
P: positive examples (ground facts)
N: negative examples (ground facts)

Find a hypothesis H such that

H "covers" every positive example given B

$\forall p \in P: B \cup H \models p$

H does not "cover" any negative example given B

$\forall n \in N: B \cup H \not\models n$

Inductive reasoning: *relation to abduction*

in inductive reasoning, the hypothesis (what has to be added to the logic program) is a set of clauses rather than a set of ground facts

given a theory T and an observation O, find an explanation E such that $T \cup E \models O$

Try to adapt the abductive meta-interpreter: inducible/1 defines the set of possible hypothesis

```
induce(E,H) :-
  induce(E, [],H).
induce(true,H,H).
induce((A,B),H0,H) :-
  induce(A,H0,H1),
  induce(B,H1,H).
induce(A,H0,H) :-
  clause(A,B),
  induce(B,H0,H).
```

```
induce(A,H0,H) :-
  element((A:-B),H0),
  induce(B,H0,H).
induce(A,H0, [(A:-B)|H]) :
  inducible((A:-B)),
  not(element((A:-B),H0)),
  induce(B,H0,H).
```

clause already assumed

assume clause if it's an inducible and not yet assumed

Inductive reasoning: *relation to abduction*

```
bird(tweety).
has_feathers(tweety).
bird(polly).
has_beak(polly).
```

```
inducible((flies(X):-bird(X),has_feathers(X),has_beak(X))).
inducible((flies(X):-has_feathers(X),has_beak(X))).
inducible((flies(X):-bird(X),has_beak(X))).
inducible((flies(X):-bird(X),has_feathers(X))).
inducible((flies(X):-bird(X))).
inducible((flies(X):-has_feathers(X))).
inducible((flies(X):-has_beak(X))).
inducible((flies(X):-true)).
```

enumeration of possible hypotheses

probably an overgeneralization

```
?-induce(flies(tweety),H).
H = [(flies(tweety):-bird(tweety),has_feathers(tweety))];
H = [(flies(tweety):-bird(tweety))];
H = [(flies(tweety):-has_feathers(tweety))];
H = [(flies(tweety):-true)];
No more solutions
```

Listing all inducible hypothesis is impractical. Better to **systematically search** the **hypothesis space** (typically large and possibly infinite when functors are involved).

Avoid overgeneralization by including **negative examples** in search process.

Inductive reasoning:

a hypothesis search involving successive generalization and specialization steps of a current hypothesis

ground fact for the predicate of which a definition is to be induced that is either true (+ example) or false (- example) under the intended interpretation

example	action	hypothesis
+ p(b, [b])	add clause	p(X,Y).
- p(x, [])	specialize	p(X, [V|W]).
- p(x, [a,b])	specialize	p(X, [X|W]).
+ p(b, [a,b])	add clause	p(X, [X|W]). p(X, [V|W]):-p(X,W).

this negative example precludes the previous hypothesis' second argument from unifying with the empty list

Generalizing clauses:
θ-subsumption

c1 is more general than c2

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that c1θ ⊆ c2

H1;...;Hn :- B1,...,Bm
H1 ∨...∨ Hn ∨ ¬B1 ∨...∨ ¬Bm

clauses are seen as sets of disjuncted positive (head) and negative (body) literals

```
element(X,V) :- element(X,Z)
```

θ-subsumes

```
element(X,[Y|Z]) :- element(X,Z)
```

using θ = {V → [Y|Z]}

```
a(X) :- b(X)
```

θ-subsumes

```
a(X) :- b(X), c(X).
```

using θ = id

Generalizing clauses:
θ-subsumption versus ⊨

H1 is at least as general as H2 given B ⇔

H1 covers everything covered by H2 given B
∀ p ∈ P: B ∪ H2 ⊨ p ⇒ B ∪ H1 ⊨ p

B ∪ H1 ⊨ H2

clause c1 θ-subsumes c2 ⇒ c1 ⊨ c2

The reverse is not true:

```
a(X) :- b(X). % c1
p(X) :- p(X). % c2
```

c1 ⊨ c2, but there is no substitution θ such that c1θ ⊆ c2

Generalizing clauses:
testing for θ-subsumption

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that c1θ ⊆ c2

no variables substituted by θ in c2:
testing for θ-subsumption amounts to testing for subset relation (allowing unification) between a ground version of c2 and c1

```
theta_subsumes((H1:-B1),(H2:-B2)):-
  verify((ground((H2:-B2)),H1=H2,subset(B1,B2))).

verify(Goal) :-
  not(not(call(Goal))).

ground(Term):-
  numbervars(Term,0,N).
```

prove Goal, but without creating bindings

Generalizing clauses:
testing for θ-subsumption

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that c1θ ⊆ c2

bodies are lists of atoms

```
?- theta_subsumes((element(X,V):- []),
                  (element(X,V):- [element(X,Z)])).
yes.

?- theta_subsumes((element(X,a):- []),
                  (element(X,V):- [])).
no.
```

Generalizing clauses:
generalizing 2 atoms

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that c1θ ⊆ c2

a1 `element(1, [1]).`

`element(z, [z,y,x]).` a2

subsumes using θ = {X/1, Y/[]}

subsumes using θ = {X/z, Y/[y,x]}

a3 `element(X, [X|Y]).`

first element of second argument (a non-empty list) has to be the first argument

happens to be the **least general** (or most specific) **generalization**
because all other atoms that θ-subsume a1 and a2 also θ-subsume a3:

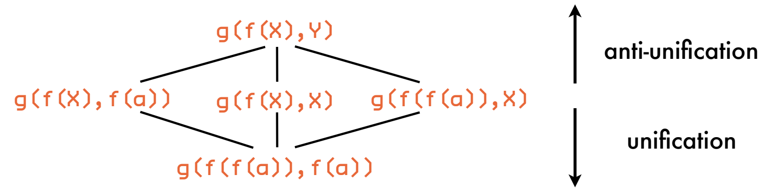
`element(X, [Y|Z]).`

only requires second argument to be an arbitrary non-empty list

`element(X,Y).`

no restrictions on either argument

Generalizing clauses:
generalizing 2 atoms - set of first-order terms is a lattice



t1 is more general than t2 ⇔ for some substitution θ: t1θ = t2

greatest lower bound of two terms (meet operation): unification

specialization = applying a substitution

least upper bound of two terms (join operation): **anti-unification**

generalization = applying an inverse substitution (terms to variables)

Generalizing clauses:
anti-unification computes the least-general generalization of two atoms under θ-subsumption

dual of unification

compare corresponding argument terms of two atoms,
replace by variable if they are different
replace subsequent occurrences of same term by same variable

θ-LGG of first two arguments

remaining arguments: inverse substitutions for each term and their accumulators

```
?- anti_unify(2*2=2+2,2*3=3+3,T,[],S1,[],S2).
T = 2*X=X+X
S1 = [2 <- X]
S2 = [3 <- X]
```

will not compute proper inverse substitutions: not clear which occurrences of 2 are mapped to X (all but the first)
BUT we are only interested in the θ-LGG

clearly, Prolog will generate a new anonymous variable (e.g., _G123) rather than X

Generalizing clauses:
anti-unification computes the least-general generalization of two atoms under θ-subsumption

```
:- op(600,xfx,'<-').
anti_unify(Term1,Term2,Term) :-
  anti_unify(Term1,Term2,Term,[],S1,[],S2).
anti_unify(Term1,Term2,Term1,S1,S1,S2,S2) :-
  Term1 == Term2,
  !.
anti_unify(Term1,Term2,V,S1,S1,S2,S2) :-
  subs_lookup(S1,S2,Term1,Term2,V),
  !.
anti_unify(Term1,Term2,Term,S10,S1,S20,S2) :-
  nonvar(Term1),
  nonvar(Term2),
  functor(Term1,F,N),
  functor(Term2,F,N),
  !,
  functor(Term,F,N),
  anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2).
anti_unify(Term1,Term2,V,S10,[Term1<-V|S10],S20,[Term2<-V|S20]).
```

same terms

not the same terms, but each has already been mapped to the same variable V in the respective inverse substitutions

equivalent compound term is constructed if both original compounds have the same functor and arity

if all else fails, map both terms to the same variable

Generalizing clauses:

anti-unification computes the least-general generalization of two atoms under θ-subsumption

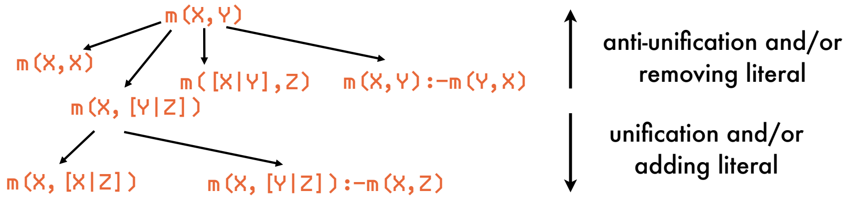
```
anti_unify_args(0,Term1,Term2,Term,S1,S1,S2,S2).
anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2):-
  N>0,
  N1 is N-1,
  arg(N,Term1,Arg1),
  arg(N,Term2,Arg2),
  arg(N,Term,ArgN),
  anti_unify(Arg1,Arg2,ArgN,S10,S11,S20,S21),
  anti_unify_args(N1,Term1,Term2,Term,S11,S1,S21,S2).
```

anti-unify first N corresponding arguments

```
subs_lookup([T1<-V|Subs1],[T2<-V|Subs2],Term1,Term2,V) :-
  T1 == Term1,
  T2 == Term2,
  !.
subs_lookup([S1|Subs1],[S2|Subs2],Term1,Term2,V):-
  subs_lookup(Subs1,Subs2,Term1,Term2,V).
```


Generalizing clauses:

set of (equivalence classes of) clauses is a lattice



C1 is more general than C2 ⇔ for some substitution θ: C1θ ⊆ C2

greatest lower bound of two clauses (meet operation): θ-MGS

specialization = applying a substitution and/or adding a literal

least upper bound of two clauses (join operation): θ-LGG

generalization = applying an inverse substitution and/or removing a literal

Generalizing clauses:

computing the θ least-general generalization

similar to, and depends on, anti-unification of atoms

but the body of a clause is (declaratively spoken) unordered

therefore have to compare all possible pairs of atoms (one from each body)

```
?- theta_lgg((element(c,[b,c]):-[element(c,[c])]),
             (element(d,[b,c,d]):-[element(d,[c,d]),element(d,[d])]),
             C).
C = element(X,[b,c|Y]):-[element(X,[c|Y]),element(X,[X])]
```

obtained by anti-unifying original heads

obtained by anti-unifying `element(c,[c])` and `element(d,[c,d])`

obtained by anti-unifying `element(c,[c])` and `element(d,[d])`

Generalizing clauses:

computing the θ least-general generalization

```
theta_lgg((H1:-B1),(H2:-B2),(H:-B)):-
  anti_unify(H1,H2,H,[],S10,[],S20),
  theta_lgg_bodies(B1,B2,[],B,S10,S1,S20,S2).
```

anti-unify heads

pair-wise anti-unification of atoms in bodies

```
theta_lgg_bodies([],B2,B,B,S1,S1,S2,S2).
theta_lgg_bodies([Lit|B1],B2, B0,B, S10,S1, S20,S2):-
  theta_lgg_literal(Lit,B2, B0,B00, S10,S11, S20,S21),
  theta_lgg_bodies(B1,B2, B00,B, S11,S1, S21,S2).
```

atom from first body

```
theta_lgg_literal(Lit1,[], B,B, S1,S1, S2,S2).
theta_lgg_literal(Lit1,[Lit2|B2],B0,B,S10,S1,S20,S2):-
  same_predicate(Lit1,Lit2),
  anti_unify(Lit1,Lit2,Lit,S10,S11,S20,S21),
  theta_lgg_literal(Lit1,B2,[Lit|B0],B, S11, S1,S21,S2).
theta_lgg_literal(Lit1,[Lit2|B2],B0,B,S10,S1,S20,S2):-
  not(same_predicate(Lit1,Lit2)),
  theta_lgg_literal(Lit1,B2,B0,B,S10,S1,S20,S2).
same_predicate(Lit1,Lit2) :-
  functor(Lit1,P,N),
  functor(Lit2,P,N).
```

atom from second body

incompatible pair

Generalizing clauses:

computing the θ least-general generalization

```
?- theta_lgg((reverse([2,1], [3], [1,2,3]):-[reverse([1], [2,3], [1,2,3])]),
             (reverse([a], [], [a]):-[reverse([], [a], [a])]),
             C).
C = reverse([X|Y], Z,  [U|V]) :-  [reverse(Y,  [X|Z],  [U|V])]
```

```
rev([2,1],[3],[1,2,3]):-rev([1],[2,3],[1,2,3])
     | |   |    |  /         |   |  |   |  /
     X Y   Z    U V          Y   X  Z   U V
     | /   |    | /          |   | /    | /
rev([a]  ,[] ,[a]    ):-rev([] ,[a]  ,[a]    )
```


Bottom-up induction:

specific-to-general search of the hypothesis space

generalizes positive examples into a hypothesis
rather than specializing the most general hypothesis as long as it covers negative examples

relative least general generalization **rlgg(e1,e2,M)**
of two positive examples e1 and e2
relative to a partial model M is defined as:
rlgg(e1, e2, M) = lgg((e1 :- Conj(M)), (e2 :- Conj(M)))

conjunction of all positive examples plus ground facts for the background predicates

Bottom-up induction:

relative least general generalization

M

```
e1  append([1,2], [3,4], [1,2,3,4]).
e2  append([a], [], [a]).
    append([], [], []).
    append([2], [3,4], [2,3,4]).
```

rlgg(e1,e2,M)

```
?- theta_lgg((append([1,2], [3,4], [1,2,3,4]) :-
                 [append([1,2], [3,4], [1,2,3,4]),
                  append([a], [], [a]), append([], [], []),
                  append([2], [3,4], [2,3,4])]),
             (append([a], [], [a]):-
                 [append([1,2], [3,4], [1,2,3,4]),
                  append([a], [], [a]),append([], [], []),
                  append([2], [3,4], [2,3,4])]),
             C)
```


Bottom-up induction:

relative least general generalization - need for pruning

rlgg(e1,e2,M)

```
append([X|Y], Z,  [X|U]) :- [
  append([2],  [3, 4],  [2, 3, 4]),
  append(Y, Z, U),
  append([V],  Z,  [V|Z]),
  append([K|L],  [3, 4],  [K, M, N|O]),
  append(L, P, Q),
  append([],  [],  []),
  append(R,  [],  R),
  append(S,  P,  T),
  append([A],  P,  [A|P]),
  append(B,  [],  B),
  append([a],  [],  [a]),
  append([C|L],  P,  [C|Q]),
  append([D|Y],  [3, 4],  [D, E, F|G]),
  append(H,  Z,  I),
  append([X|Y],  Z,  [X|U]),
  append([1, 2],  [3, 4],  [1, 2, 3, 4])
]
```

remaining ground facts from M (e.g., examples) are redundant: can be removed

introduces variables that do not occur in the head: can assume that hypothesis clauses are constrained

head of clause in body = tautology: restrict ourselves to strictly constrained hypothesis clauses

variables in body are **proper** subset of variables in head

Bottom-up induction:
relative least general generalization - algorithm

to determine vars in head (strictly constrained restriction)

```
rlgg(E1,E2,M,(H:- B)):-
  anti_unify(E1,E2,H,[],S10,[],S20),
  varsin(H,V),
  rlgg_bodies(M,M,[],B,S10,S1,S20,S2,V).
```

`rlgg_bodies(B0,B1,BR0,BR,S10,S1,S20,S2,V)`: rlgg all literals in B0 with all literals in B1, yielding BR (from accumulator BR0) containing only vars in V

```
rlgg_bodies([],B2,B,B,S1,S1,S2,S2,V).
rlgg_bodies([L|B1],B2,B0,B,S10,S1,S20,S2,V):-
  rlgg_literal(L,B2,B0,B00,S10,S11,S20,S21,V),
  rlgg_bodies(B1,B2,B00,B,S11,S1,S21,S2,V).
```


Bottom-up induction:
relative least general generalization - algorithm

```
rlgg_literal(L1,[],B,B,S1,S1,S2,S2,V).
rlgg_literal(L1,[L2|B2],B0,B,S10,S1,S20,S2,V):-
  same_predicate(L1,L2),
  anti_unify(L1,L2,L,S10,S11,S20,S21),
  varsin(L,Vars),
  var_proper_subset(Vars,V),
  !,
  rlgg_literal(L1,B2,[L|B0],B,S11,S1,S21,S2,V).
rlgg_literal(L1,[L2|B2],B0,B,S10,S1,S20,S2,V):-
  rlgg_literal(L1,B2,B0,B,S10,S1,S20,S2,V).
```

strictly constrained (no new variables, but proper subset)

otherwise, an incompatible pair of literals

Bottom-up induction:
relative least general generalization - algorithm

```
var_proper_subset([],Ys) :-
  Ys \= [].
var_proper_subset([X|Xs],Ys) :-
  var_remove_one(X,Ys,Zs),
  var_proper_subset(Xs,Zs).
```

```
var_remove_one(X,[Y|Ys],Ys) :-
  X == Y.
var_remove_one(X,[Y|Ys],[Y|Zs) :-
  var_remove_one(X,Ys,Zs).
```

```
varsin(Term,Vars):-
  varsin(Term,[],V),
  sort(V,Vars).
varsin(V,Vars,[V|Vars]):-
  var(V).
varsin(Term,V0,V):-
  functor(Term,F,N),
  varsin_args(N,Term,V0,V).
```

```
varsin_args(0,Term,Vars,Vars).
varsin_args(N,Term,V0,V):-
  N>0,
  N1 is N-1,
  arg(N,Term,ArgN),
  varsin(ArgN,V0,V1),
  varsin_args(N1,Term,V1,V).
```


Bottom-up induction:
relative least general generalization - algorithm

```
?- rlgg(append([1,2],[3,4],[1,2,3,4]),
        append([a],[],[a]),
        [append([1,2],[3,4],[1,2,3,4]),
         append([a],[],[a]),
         append([],[],[]),
         append([2],[3,4],[2,3,4])],
        (H:- B)).
H = append([X|Y], Z, [X|U])
B = [append([2], [3, 4], [2, 3, 4]),
     append(Y, Z, U),
     append([], [], []),
     append([a], [], [a]),
     append([1, 2], [3, 4], [1, 2, 3, 4])]
```

Bottom-up induction:
main algorithm

construct rlgg of two positive examples
- remove all positive examples that are extensionally covered by the constructed clause

further generalize the clause by removing literals
- as long as no negative examples are covered

Bottom-up induction:
main algorithm

```
induce_rlgg(Exs,Clauses):-
  pos_neg(Exs,Poss,Negs),
  bg_model(BG),
  append(Poss,BG,Model),
  induce_rlgg(Poss,Negs,Model,Clauses).

induce_rlgg(Poss,Negs,Model,Clauses):-
  covering(Poss,Negs,Model,[],Clauses).
```

split positive from negative examples

include positive examples in background model

```
pos_neg([],[],[]).
pos_neg([+E|Exs],[E|Poss],Negs):-
  pos_neg(Exs,Poss,Negs).
pos_neg([-E|Exs],Poss,[E|Negs]):-
  pos_neg(Exs,Poss,Negs).
```


Bottom-up induction:
main algorithm - covering

```
covering(Poss,Negs,Model,Hyp0,NewHyp) :-
  construct_hypothesis(Poss,Negs,Model,Hyp),
  !,
  remove_pos(Poss,Model,Hyp,NewPoss),
  covering(NewPoss,Negs,Model,[Hyp|Hyp0],NewHyp).
covering(P,N,M,H0,H) :-
  append(H0,P,H).
```

construct a new hypothesis clause that covers all of the positive examples and none of the negative

remove covered positive examples

when no longer possible to construct new hypothesis clauses, add remaining positive examples to hypothesis

```
remove_pos([],M,H,[]).
remove_pos([P|Ps],Model,Hyp,NewP) :-
  covers_ex(Hyp,P,Model),
  !,
  write('Covered example: '),
  write_ln(P),
  remove_pos(Ps,Model,Hyp,NewP).
remove_pos([P|Ps],Model,Hyp,[P|NewP]):-
  remove_pos(Ps,Model,Hyp,NewP).
```

```
covers_ex((Head:- Body),
          Example,Model):-
verify((Head=Example,
       forall(element(L,Body),
              element(L,Model)))).
```


Bottom-up induction:
main algorithm - hypothesis construction

```
construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
  write('RLGG of '), write(E1),
  write(' and '), write(E2), write(' is'),
  rlgg(E1,E2,Model,Cl),
  reduce(Cl,Negs,Model,Clause),
  !,
  nl,tab(5), write_ln(Clause).
construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
  write_ln(' too general'),
  construct_hypothesis([E2|Es],Negs,Model,Clause).
```

this is the only step in the algorithm that involves negative examples!

remove redundant literals and ensure that no negative examples are covered

if no rlgg can be constructed for these two positive examples or the constructed one covers a negative example

note that E1 will be considered again with another example in a different iteration of covering/5

Bottom-up induction:
main algorithm - hypothesis reduction

```
setof0(X,G,L):-
    setof(X,G,L),!.
setof0(X,G, []).
```

succeeds with empty list of no solutions can be found

remove redundant literals and ensure that no negative examples are covered

```
reduce((H:-B0),Negs,M,(H:-B)):-
  setof0(L,
        (element(L,B0), not(var_element(L,M))),
        B1),
  reduce_negs(H,B1,[],B,Negs,M).
```

removes literals from the body that are already in the model

```
var_element(X,[Y|Ys]):-
  X == Y.
var_element(X,[Y|Ys]):-
  var_element(X,Ys).
```

element/2 using syntactic identity rather than unification

Bottom-up induction:
main algorithm - hypothesis reduction

B is the body of the reduced clause: a subsequence of the body of the original clause (second argument), such that no negative example is covered by model U reduced clause (H:-B)

```
reduce_negs(H,[L|Rest],B0,B,Negs,Model):-
  append(B0,Rest,Body),
  not(covers_neg((H:-Body),Negs,Model,N)),
  !,
  reduce_negs(H,Rest,B0,B,Negs,Model).
reduce_negs(H,[L|Rest],B0,B,Negs,Model):-
  reduce_negs(H,Rest,[L|B0],B,Negs,Model).
reduce_negs(H,[],Body,Body,Negs,Model):-
  not(covers_neg((H:- Body),Negs,Model,N)).
```

try to remove L from the original body

L cannot be removed

fail if the resulting clause covers a negative example

```
covers_neg(Clause,Negs,Model,N) :-
  element(N,Negs),
  covers_ex(Clause,N,Model).
```

a negative example is covered by clause U model

Bottom-up induction:
example

```
?- induce_rlgg([
+append([1,2],[3,4],[1,2,3,4]),
+append([a],[],[a]),
+append([],[],[]),
+append([],[1,2,3],[1,2,3]),
+append([2],[3,4],[2,3,4]),
+append([],[3,4],[3,4]),
-append([a],[b],[b]),
-append([c],[b],[c,a]),
-append([1,2],[],[1,3])
], Clauses).
```

```
RLGG of append([1,2],[3,4],[1,2,3,4]) and append([a],[],[a]) is
append([X|Y],Z,[X|U]) :- [append(Y,Z,U)]
Covered example: append([1,2],[3,4],[1,2,3,4])
Covered example: append([a],[],[a])
Covered example: append([2],[3,4],[2,3,4])

RLGG of append([],[],[]) and append([],[1,2,3],[1,2,3]) is
append([],X,X) :- []
Covered example: append([],[],[])
Covered example: append([],[1,2,3],[1,2,3])
Covered example: append([],[3,4],[3,4])

Clauses = [(append([],X,X) :- []),
(append([X|Y],Z,[X|U]) :- [append(Y,Z,U)])]
```


Bottom-up induction:
example

```
bg_model([num(1,one),num(2,two),
          num(3,three),
          num(4,four),
          num(5,five)]).
?-induce_rlgg([
+listnum([],[]),
+listnum([2,three,4],[two,3,four]),
+listnum([4],[four]),
+listnum([three,4],[3,four]),
+listnum([two],[2]),
-listnum([1,4],[1,four]),
-listnum([2,three,4],[two]),
-listnum([five],[5,5]) ],
Clauses).
```

```
RLGG of listnum([],[]) and
        listnum([2,three,4],[two,3,four]) is too general
RLGG of listnum([2,three,4],[two,3,four]) and
        listnum([4],[four]) is
listnum([X|Xs],[Y|Ys]):-[num(X,Y),listnum(Xs,Ys)]
Covered example: listnum([2,three,4],[two,3,four])
Covered example: listnum([4],[four])
RLGG of listnum([],[]) and listnum([three,4],[3,four]) is too general
RLGG of listnum([three,4],[3,four]) and listnum([two],[2]) is
listnum([V|Vs],[W|Ws]):-[num(W,V),listnum(Vs,Ws)]
Covered example:
listnum([three,4],[3,four])
Covered example: listnum([two],[2])
Clauses =[(listnum([V|Vs],[W|Ws]):-[num(W,V),listnum(Vs,Ws)]),
    (listnum([X|Xs],[Y|Ys]):-[num(X,Y),listnum(Xs,Ys)]),listnum([],[]) ]
```

programming with quantified truth

programming with qualified truth

programming with constraints on integer domains

Declarative Programming

8: interesting loose ends

only to whet your appetite, will **not** be asked on exam

implicit parallel evaluation

software engineering applications

Logic programming with quantified truth:

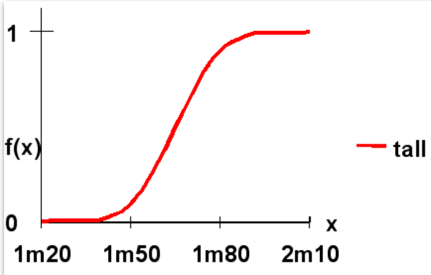
reasoning with vague (rather than incomplete) information

fuzzy set [Zadeh 1965]

characteristic function generalised to allow gradual membership

$$\mu_A : U \longrightarrow [0,1]$$

$$\mu_A(x) = \begin{cases} 0 \leftrightarrow x \notin A \\ 1 \leftrightarrow x \in A \\ 0 < \alpha < 1 \leftrightarrow x \in A \text{ to the extent } \alpha \end{cases}$$



Logic programming with quantified truth:

operations on fuzzy sets

classical set-theoretic operations

- Intersection: $\mu_{A \cap B}(x) = min(\mu_A(x), \mu_B(x))$
- Union: $\mu_{A \cup B}(x) = max(\mu_A(x), \mu_B(x))$
- Complement: $\mu_{\bar{A}}(x) = 1 - \mu_A(x)$

original ones by Zadeh, later generalized

linguistic hedges

take a fuzzy set (e.g., set of tall people) and modify its membership function

modelling adverbs: very, somewhat, indeed

compositional rule of inference

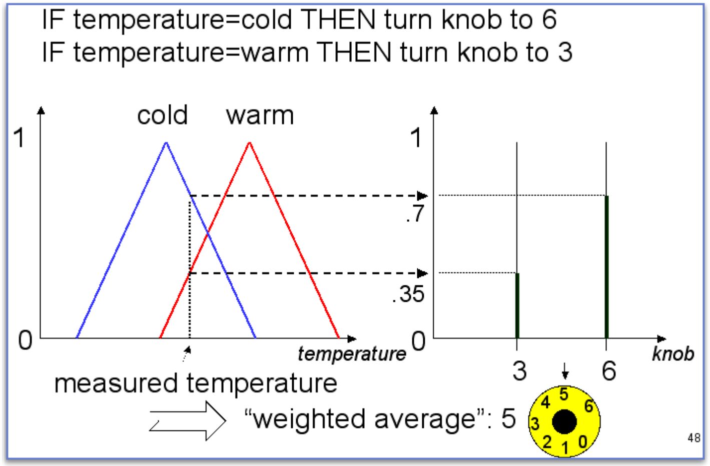
premise	if X is A and Y is B then Z is C
fact	X is A' and Y is B'
consequence	Z is C'

Logic programming with quantified truth:

killer application: fuzzy process control

Logic programming with quantified truth:

killer application: fuzzy process control



easier and smoother operation than classical process control

Logic programming with quantified truth:

killer application: fuzzy process control

$rule_1$	if X is A_1 then Y is B_1
$rule_2$	if X is A_2 then Y is B_2
...	...
fact	X is A
consequence	Y is B

Designing a fuzzy control system generally consists of the following steps:

Fuzzification This is the basic step in which one has to determine appropriate fuzzy membership functions for the input and output fuzzy sets and specify the individual rules regulating the system.

Inference This step comprises the calculation of output values for each rule even when the premises match only partially with the given input.

Composition The output of the individual rules in the rule base can now be combined into a single conclusion.

Defuzzification The fuzzy conclusion obtained through inference and composition often has to be converted to a crisp value suited for driving the motor of an air conditioning system, for example.

Logic programming with quantified truth:

a meta-interpreter for a fuzzy logic programming language

many variations possible

confidence in conclusion q given absolute truth of $q_1, \ldots, q_n$

LP with quantified truth

weighted logic rules

```
q : c if q1,...,qn where c ∈ ]0,1]
```

fuzzy resolution procedure

```
T(q) = c * min(T(q1),...,T(qn))
```

similar to f-Prolog [1990:liu]

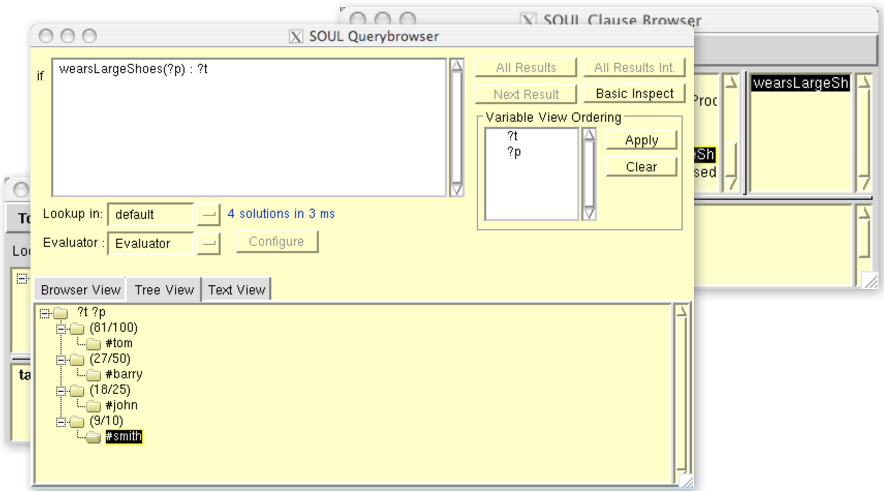
```
if popular_product(?p) : ?c
```

?p	?c
flowers	1
chips	min(0.9, 0.6)*0.8 = 0.48

```
sold(flowers, 15).
attractive_packaging(chips) : 0.9.
well_advertised(chips) : 0.6.

popular_product(?product) if
  sold(?product, ?amount),
  ?amount > 10.

popular_product(?product) : 0.8 if
  attractive_packaging(?product),
  well_advertised(?product).
```


Logic programming with quantified truth:

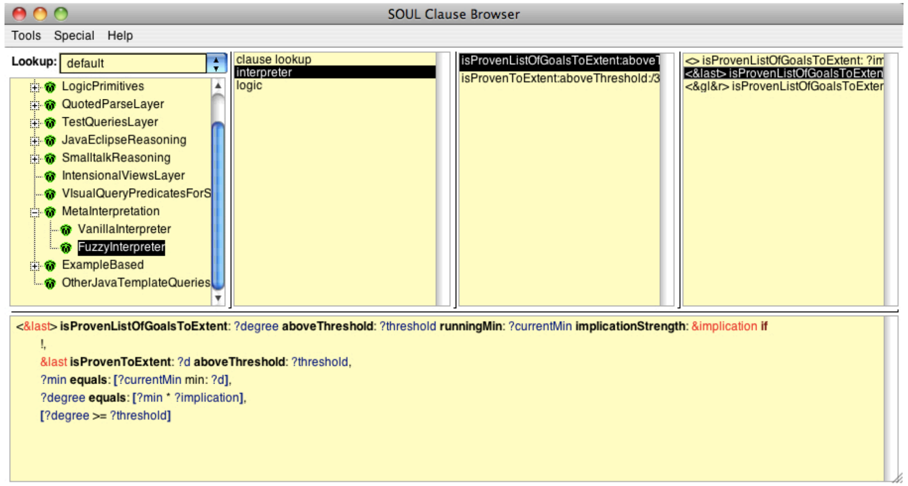
a meta-interpreter for a fuzzy logic programming language



DEMO

Logic programming with quantified truth:

a meta-interpreter for a fuzzy logic programming language



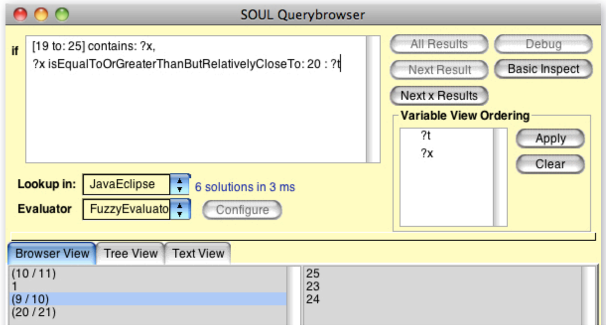
DEMO

Logic programming with quantified truth:

reifying the characteristic function of a fuzzy set

```
+?x isEqualToOrGreaterThanButRelativelyCloseTo: +?x.
+?x isEqualToOrGreaterThanButRelativelyCloseTo: +?y : ?c if
  [?x > ?y],
  ?c equals: [(?y / ?x) max: (9 / 10)]
```

associates a truth degree
[9,1[
with numbers ?x that are
greater than ?y, but do not
deviate more than 10% from ?y



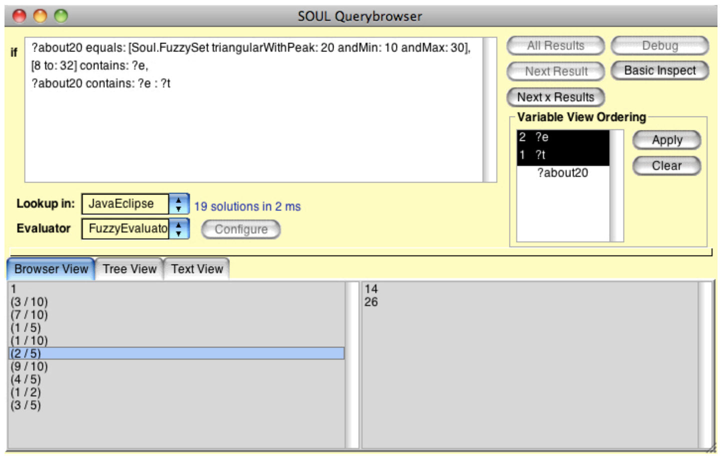
DEMO

Logic programming with quantified truth:

quantifying over the elements of a fuzzy set

```
+?c contains: +?e if
  [?c isKindOf: Soul.FuzzySet],
  [?c membershipDegreeOfElement: ?e]
```

additional contains:/2
clause for fuzzy sets
implemented in Smalltalk



linearly models
how close an
element is to 20

$$\Delta(x,\alpha,\beta,\gamma) = \begin{cases} 0 & x < \alpha \\ (x-\alpha)/(\beta-\alpha) & \alpha \le x \le \beta \\ (\gamma-x)/(\gamma-\beta) & \beta \le x \le \gamma \\ 0 & x > \gamma \end{cases}$$

Logic programming with qualified truth:

an executable linear temporal logic (informally)

regular logic formulas qualified by temporal operators:

- □ (always)
- ◇ (sometimes)
- ● (previous)
- ○ (next).

evaluated against an implicit temporal context:

□ϕ is true if ϕ is true at all moments in time.

we will assume a finite, non-branching timeline for our example application: reasoning about execution traces of a program

Logic programming with qualified truth:

a meta-interpreter for finite linear temporal logic programming

```
solve(A) :-
  prove(A, 0).

prove(not(A), T) :-
  not(prove(A, T)).

prove(next(A), T) :-
  NT #= T + 1,
  prove(A, NT).
prove(next(C, A), T) :-
  C #> 0,
  NT #= T + C,
  prove(A, NT).

prove(previous(A), T) :-
  NT #= T - 1,
  prove(A, NT).
prove(previous(C, A), T) :-
  C #> 0,
  NT #= T - C,
  prove(A, NT).
```

> the initial temporal context for all top-level formulas is the beginning of the timeline

> next(A) holds if A holds at the next moment in time

> next(C,A) holds if A holds C steps into the future (possibly a variable)

> #> and friends impose constraints over integer domain: use_module(library(clpfd)).

Intermezzo:

constraint logic programming over integer domains

```
?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19\/21..sup.

?- 2*X #= 10.
X = 5.

?- X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs), X = 1, Y #\= 2.
Vs = [1, 3, 2],
X = 1,
Y = 3,
Z = 2.
```

> X in integer domain

> X in union of two domains

> list of variables on the left is in the domain on the right

> ensures elements are assigned different values from domain

Intermezzo:

constraint logic programming over integer domains
SEND + MORE = MONEY

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
        all_different(Vars),
        S*1000 + E*100 + N*10 + D +
             M*1000 + O*100 + R*10 + E #=
             M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0.
```

```
?- puzzle(As+Bs=Cs).
As = [9, _G10107, _G10110, _G10113],
Bs = [1, 0, _G10128, _G10107],
Cs = [1, 0, _G10110, _G10107, _G10152],
_G10107 in 4..7,
1000*9+91*_G10107+ -90*_G10110+_G10113+ -9000*1+ -900*0+10*_G10128+ -1*_G10152#=0,
all_different([_G10107, _G10110, _G10113, _G10128, _G10152, 0, 1, 9]),
_G10110 in 5..8,
_G10113 in 2..8,
_G10128 in 2..8,
_G10152 in 2..8.
```

> deduced more stringent constraints for variables

Intermezzo:

constraint logic programming over integer domains
SEND + MORE = MONEY

```
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
  Vars = [S,E,N,D,M,O,R,Y],
  Vars ins 0..9,
  all_different(Vars),
  S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
  M #\= 0, S #\= 0.
```

```
?- puzzle(As+Bs=Cs), label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.
```

> labeling a domain variable systematically tries out values for it until it is ground

```
?- puzzle(As+Bs=Cs).
As = [9, _G10107, _G10110, _G10113],
Bs = [1, 0, _G10128, _G10107],
Cs = [1, 0, _G10110, _G10107, _G10152],
_G10107 in 4..7,
1000*9+91*_G10107+ -90*_G10110+_G10113+ -9000*1+ -900*0+10*_G10128+ -1*_G10152#=0,
all_different([_G10107, _G10110, _G10113, _G10128, _G10152, 0, 1, 9]),
_G10110 in 5..8,
_G10113 in 2..8,
_G10128 in 2..8,
_G10152 in 2..8.
```

> deduced more stringent constraints for variables

Logic programming with qualified truth:

a meta-interpreter for finite linear temporal logic programming

```
prove(sometime(C, A), T) :-
    C#>=0,
    bot(Bot),
    eot(Tot),
    NT in Bot..Tot,
    NT #>= T,
    NT #=< T+C,
    prove(A, NT).
prove(sometime(C,A), T) :-
    C #=< 0,
    bot(Bot),
    eot(Tot),
    NT in Bot..Tot,
    NT #>= T + C,
    NT #=< T,
    prove(A, NT).
prove(sometime(A), _) :-
    bot(Bot),
    eot(Tot),
    C in Bot..Tot,
    prove(A, C).
```

A holds sometime between now and C steps in the future

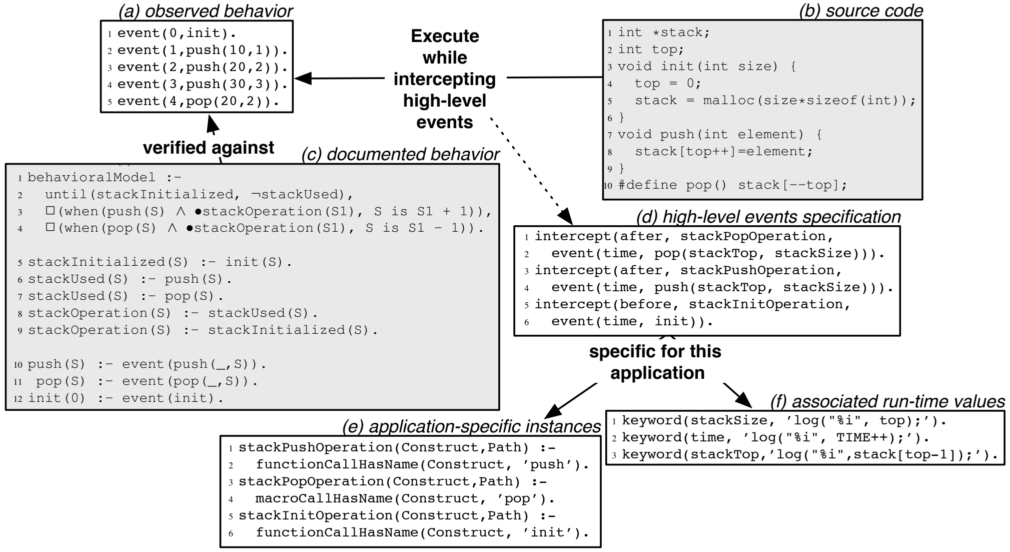
A holds sometime between now and C steps in the past

A holds somewhere on the timeline

similar for always

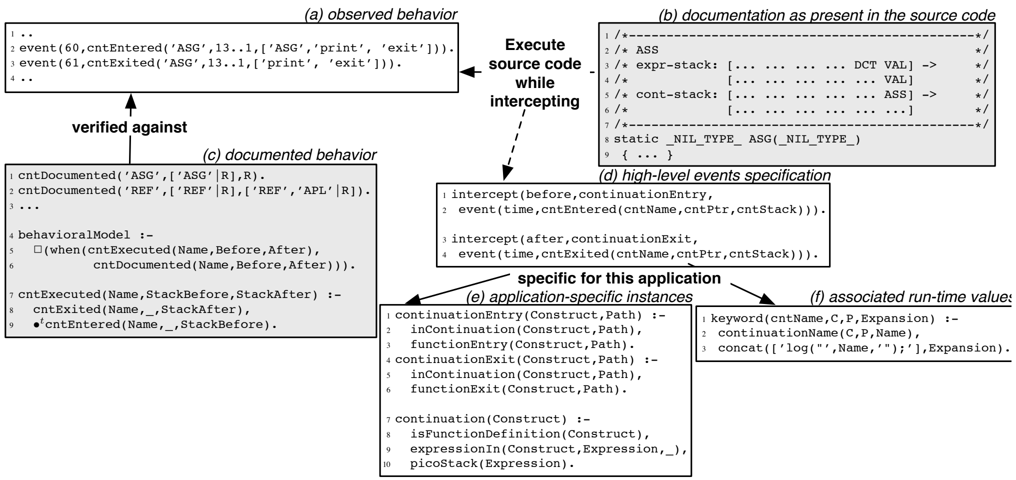
Logic programming with qualified truth:

example application: reasoning about execution traces



Logic programming with qualified truth:

example application: reasoning about execution traces



Non-standard evaluation strategies:

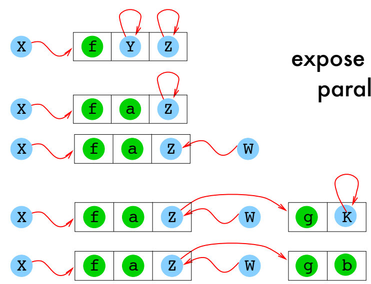
a taste of implicit parallel evaluation

multi-core revolution

speed up sequential programs

should be easier for declarative programs

```
main :- X = f(Y,Z),
        Y = a,
        W = Z,
        W = g(K),
        X = f(a,g(b)).
```



expose inherent parallelism

formal foundation

relatively pure

BUT also complex datastructures with pointers ...
imagine executing these goals in parallel!

[http://clip.dia.fi.upm.es/~logalg/slides/PS/A_par.pdf]

Non-standard evaluation strategies:

a taste of implicit parallel evaluation

while (Query not empty) **do**
  select$_{literal}$ B from Query — *And-Parallelism*
  repeat
    select$_{clause}$ (H :- Body) from Program — *Or-Parallelism*
  until (unify(H,B) or no clauses left) — *Unification Parallelism*
  if (no clauses left) **then** FAIL
  else
    σ = MostGeneralUnifier(H,B)
    Query = ((Query \ {B}) ∪ Body)σ
  endif
endwhile

not trivial: goals typically depend on each other (data and control dependency), workers need to be synchronized

correctness (same solutions as sequential)
efficiency (no slowdown, speedup)

Non-standard evaluation strategies:

a taste of implicit parallel evaluation - or-parallelism

```
p(a).
p(b).
?- p(X).
```

there is no dependency between the clauses implementing p/1

execute different branches at choice point simultaneously

relevant for search problems, generate-and-test

much easier to implement than and-parallelism

issue: maintaining a different environment per branch efficiently(e.g., sharing, copying, ...)

typical architecture:

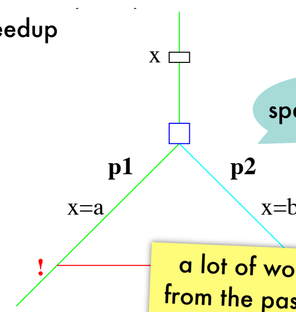
- set of workers, each a full interpreter
- scheduler assigns unexplored branches to idle workers

Non-standard evaluation strategies:

a taste of implicit parallel evaluation - or-parallelism

speculative work should be avoided to gain speedup

```
..., p(X), ...
p(X) :- ..., X=a, ..., !, ...
p(X) :- ..., X=b, ...
```



speculative

left-based scheduling, immediate killing on cut

a lot of work from the past is relevant again, BUT: distributed vs shared memory architectures, caching

```
main :- l, s.

:- parallel l/0.
l :- large_work_a.
l :- large_work_b.

:- parallel s/0.
s :- small_work_a.
s :- small_work_b.
```

avoid incurring an overhead from fine-grained parallelism

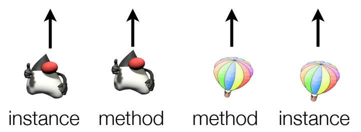
Logic programming in software engineering:

SOUL - symbiosis

symbiosis with base program languages

```
if ?c isCompilationUnit,
   [?c types size > 1]
```

← ordinary term
← symbiosis term

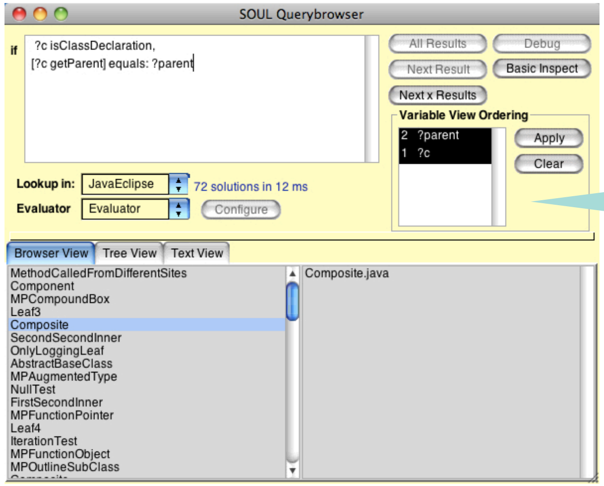


base program not **reified as logic facts**

- changes are immediately reflected
- query results easily perused by existing IDE's

Logic programming in software engineering:

SOUL - symbiosis - demo



nice, but true power of logic programming comes not only from backtracking, but also from the ability to unify with a user-provided compound term to quickly select objects one is interested in

hold that thought

hmm .. strange: the method's name (a Java Object) is unified with a compound term?

```
if ?m methodDeclarationHasName: ?n,
   ?n equals: simpleName(?identifier)
```

```
if ?m methodDeclarationHasName: simpleName(?identifier)
```


Logic programming in software engineering:

SOUL - symbiosis - demo

all subclasses of presentation.Component should define a method acceptVisitor(ComponentVisitor) that invokes System.out.println(String) before double dispatching to the argument

```
public class PrototypicalLeaf extends Component {
   public void acceptVisitor(ComponentVisitor v) {
      System.out.println("Prototypical.");
      v.visitPrototypicalLeaf(this);
   }
}
```


Logic programming in software engineering:

SOUL - symbiosis - demo

```
?type isTypeWithFullyQualifiedName: ['presentation.Component'],
?class inClassHierarchyOfType: ?type,
not(?class classDeclarationHasName: simpleName(['Composite'])),
?class definesMethod: ?m,

?m methodDeclarationHasName: simpleName(['acceptVisitor']),
?m methodDeclarationHasParameters: nodeList(<?p>),
?p singleVariableDeclarationHasName: simpleName(?id),
?m methodDeclarationHasBody: ?body,

?body equals: block(nodeList(<expressionStatement(?log),expressionStatement(?dd)>)),
or(?so equals: qualifiedName(simpleName(['System']),simpleName(['out'])),
   ?so equals: fieldAccess(simpleName(['System']),simpleName(['out']))),
?log equals: methodInvocation(?so,?,simpleName(['println']),nodeList(<?string>)),
?dd equals: methodInvocation(simpleName(?id),?,?,nodeList(<thisExpression([nil])>))
```

yuk .. not as declarative as advertised!

and I have to do this for all implementation variants?

Logic programming in software engineering:

SOUL - code templates

integrate concrete syntax of base program

```
if jtStatement(?s) {
  while(?iterator.hasNext()) {
      ?collection.add(?element);
  }
},
jtExpression(?iterator){?collection.iterator()}
```

resolved by existential queries on control-flow graph

is add(Object) ever invoked in the control-flow of a while-statement?

Logic programming in software engineering:

SOUL - code templates - demo

Logic programming in software engineering:

SOUL - code templates - demo

but still not in query results:

```
public class MustAliasLeaf extends Component {
    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("Must alias.");
        Component temp = this;
        v.visitMustAliasLeaf(temp);
    }
}
```

```
public class MayAliasLeaf extends Component {
    public Object m(Object o) {
        if(getInput() % 2 == 0)
            return o;
        else
            return new MayAliasLeaf();
    }

    public void acceptVisitor(ComponentVisitor v) {
        System.out.println("May alias.");
        v.visitMayAliasLeaf((MayAliasLeaf)m(this));
    }
}
```


Logic programming in software engineering:

SOUL - code templates - demo

```
jtClassDeclaration(?class,?interpretation){
    class !Composite extends* presentation.Component {
        ?modList ?type acceptVisitor(?t ?p) {
            System.out.println(?string);
            ?p.?m(this);
        }
    }
}
```

```
public class SuperLogLeaf extends OnlyLoggingLeaf
{
    public void acceptVisitor(ComponentVisitor v) {
        super.acceptVisitor(v);
        v.visitSuperLogLeaf(this);
    }
}
```

vs

```
?type isTypeWithFullyQualifiedName: ['presentation.Component'],
?class inClassHierarchyOfType: ?type,
not(?class classDeclarationHasName: simpleName(['Composite'])),
?class definesMethod: ?m,

?m methodDeclarationHasName: simpleName(['acceptVisitor']),
?m methodDeclarationHasParameters: nodeList(<?p>),
?p singleVariableDeclarationHasName: simpleName(?id),
?m methodDeclarationHasBody: ?body,

?body equals: block(nodeList(<expressionStatement(?log),expressionStatement(?dd)>)),
or(?so equals: qualifiedName(simpleName(['System']),simpleName(['out'])),
   ?so equals: fieldAccess(simpleName(['System']),simpleName(['out']))),
?log equals: methodInvocation(?so,?,simpleName(['println']),nodeList(<?string>)),
?dd equals: methodInvocation(simpleName(?id),?,?,nodeList(<thisExpression([nil])>))
```

Logic programming in software engineering:

SOUL - domain-specific unification

instance vs compound term

easily identify elements of interest

instance vs **instance**

incorporates static analyses: ensures query conciseness & correctness

semantic analysis
correct application of scoping rules, name resolution

points-to analysis
tolerance for syntactically differing expressions

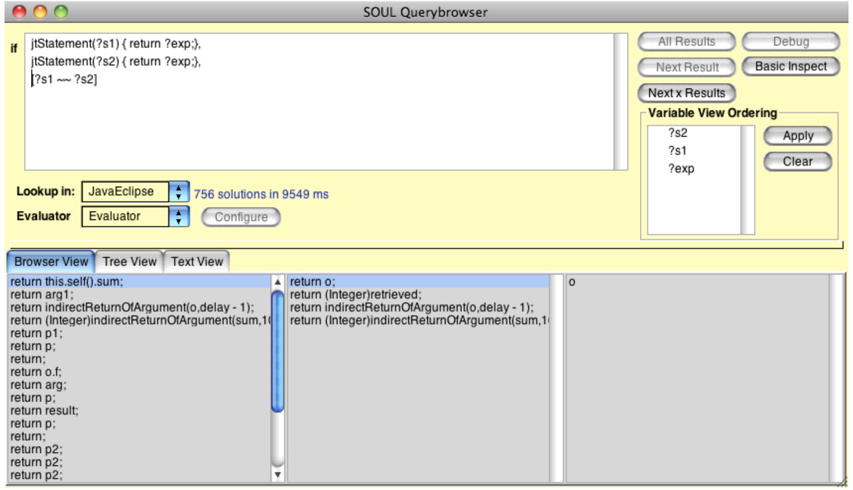
can the value on which hasNext() is invoked alias the iterator of the collection to which add is invoked?

```
if jtStatement(?s) {
  while(?iterator.hasNext()) {
    ?collection.add(?element);
  }
},
jtExpression(?iterator){?collection.iterator()}
```

never, in at least one or in all possible executions
-> propagate this knowledge using **logic of quantified truth**

Logic programming in software engineering:

SOUL - domain-specific unification - demo



Logic programming in software engineering:

SOUL - domain-specific unification - demo