Declarative Programming

1: introduction

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”

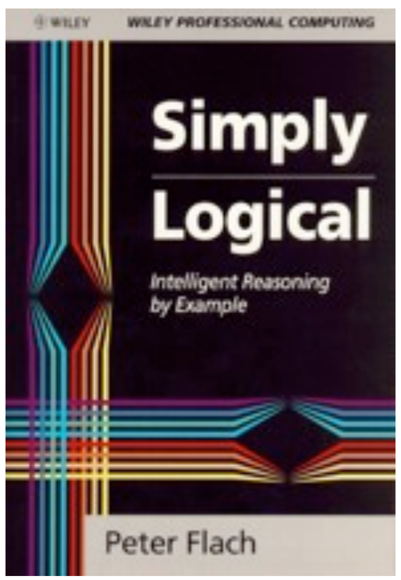
http://www.cs.bris.ac.uk/~flach/SL/slides/

slides on Computational Logic by the CLIP group

http://clip.dia.fi.upm.es/~logalg/

Practicalities

course material



exam

oral test with written preparation about theory and exercises

individual programming project

averaged, unless one ≤ 7

website

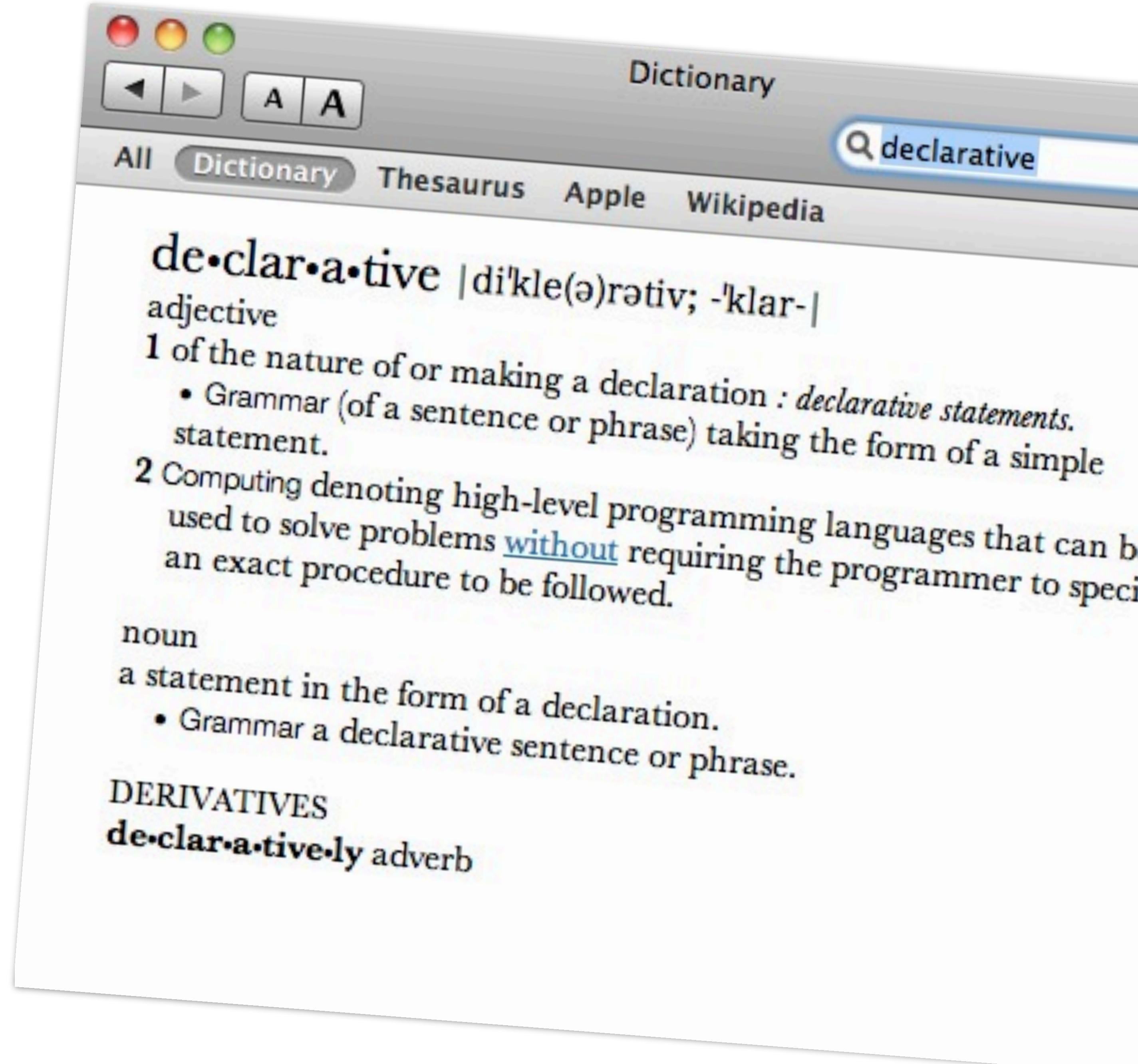
http://soft.vub.ac.be/~cderoove/declarative_programming/

exercises

5 sessions
start 6th of October at IG

Problem declaration

Problem solving strategy



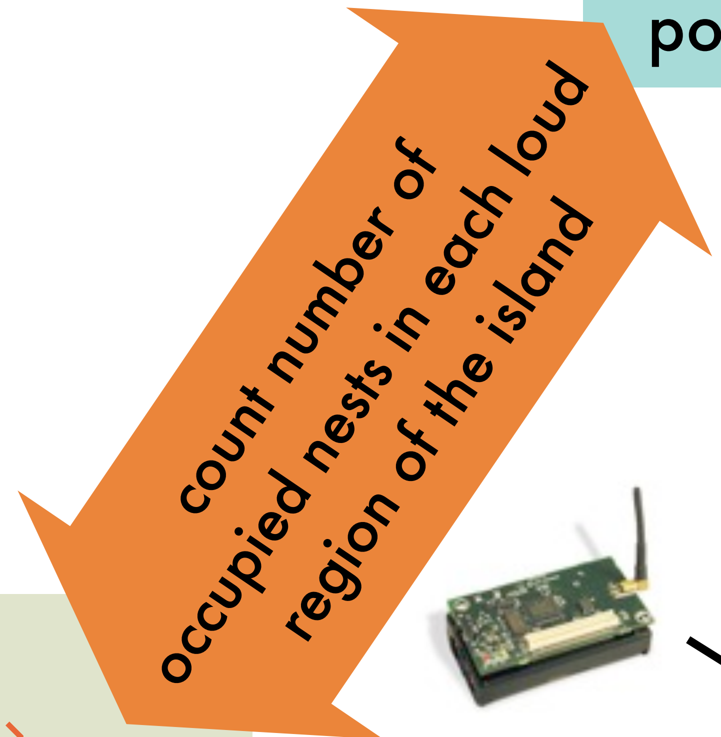
Declarative

Habitat Monitoring using Sensor Network

gather sensor readings

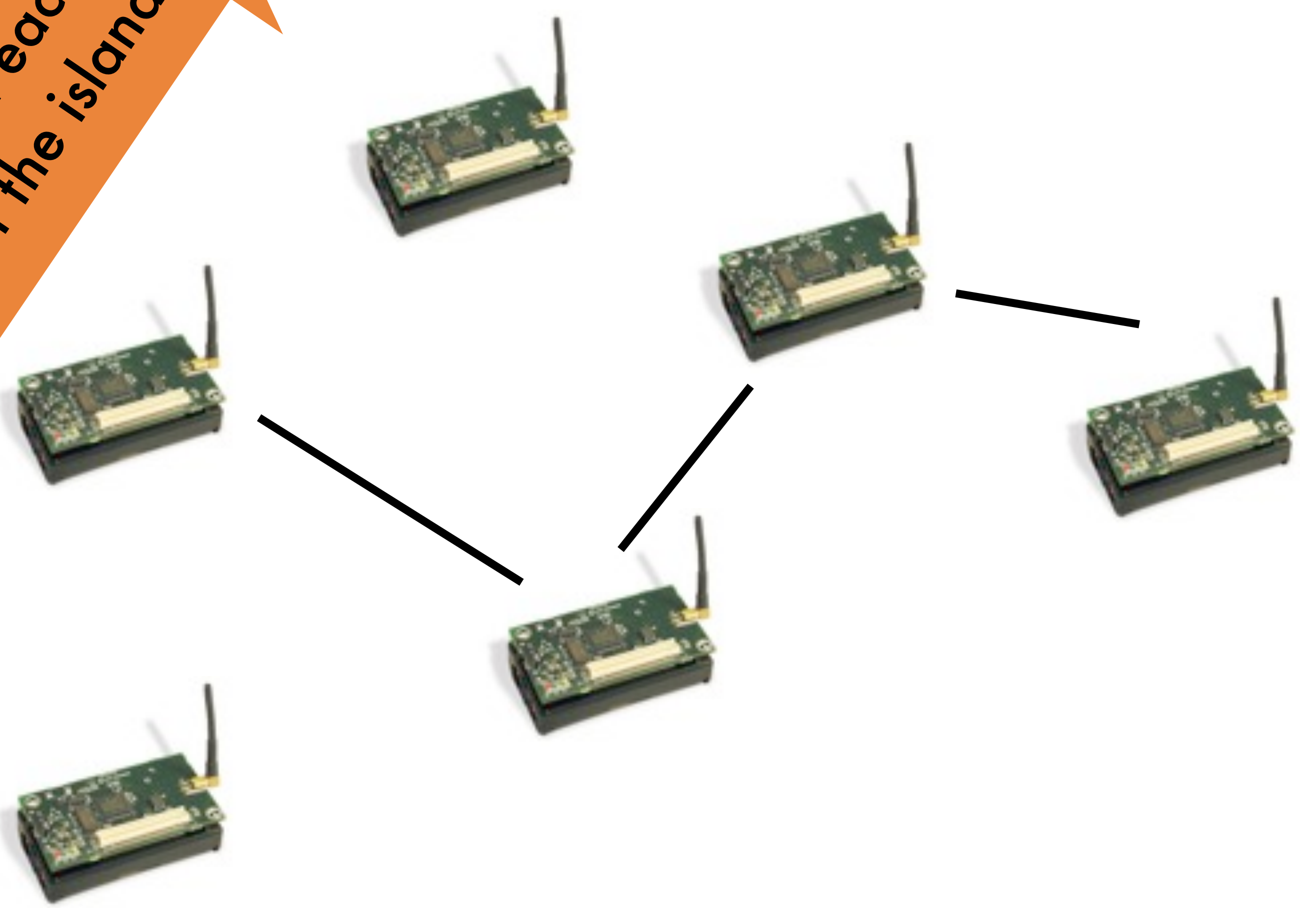
route through network while adjusting averages and count

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)]
```

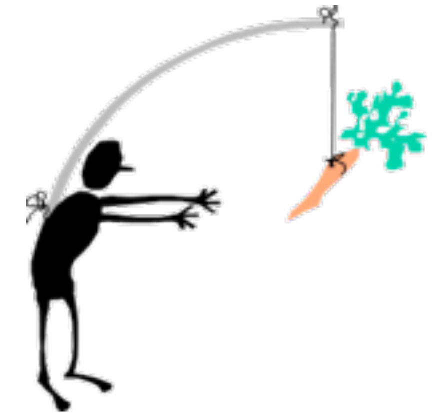
positioning GUI widgets

XWT

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

also ..

General-purpose declarative programming: logic formalizes human thought process



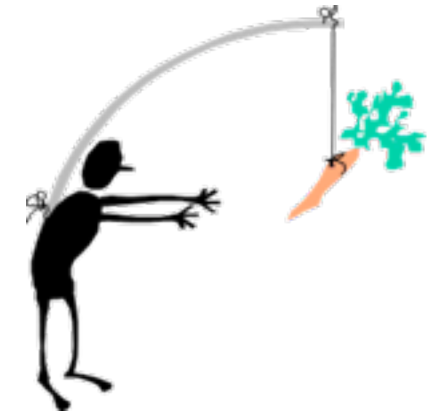
classical logic

```
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

Peano encoding natural numbers

```
nat(0) ∧ nat(s(0)) ∧ nat(s(s(0))) ∧ . . .
```

extensionally

```
nat(0) ∧
∀X : nat(X) → nat(s(X)))
```

intensionally

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))
```

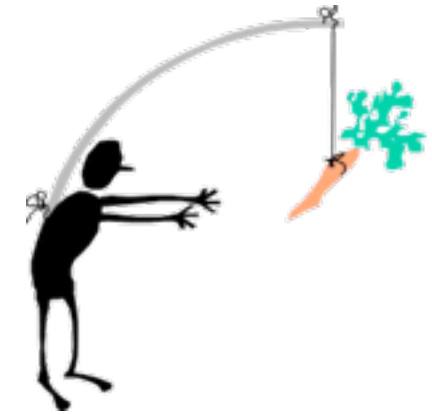
squares

```
∀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



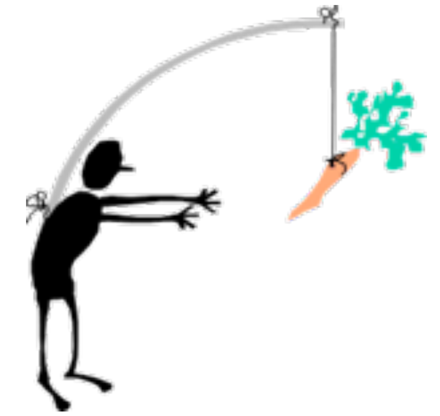
Assuming the existence of a mechanical proof procedure, a new view of problem solving and computing is possible

[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

expressivity

p versus p(X)

logics of quantified truth

logics of qualified truth

...

which proof procedure

performance

concurrency, memoization ..

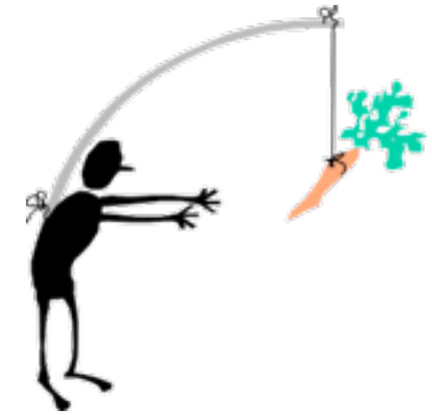
soundness

are all provables true

completeness

can all trues be proven

General-purpose declarative programming: historical overview



60

Greene: problem solving.

Robinson: linear resolution.

70

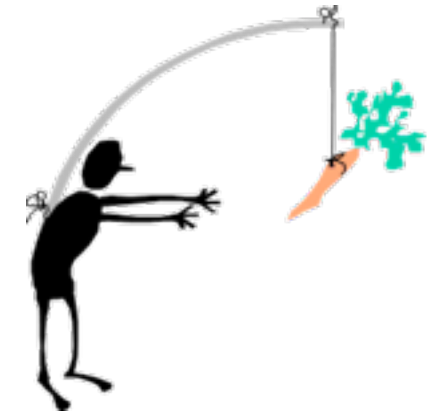
(early) Kowalski: procedural interpretation of Horn clause logic. Read:
A if B_1 and B_2 and $\cdots$ and B_n as:
to solve (execute) A, solve (execute) B_1 and B_2 and,..., B_n

(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



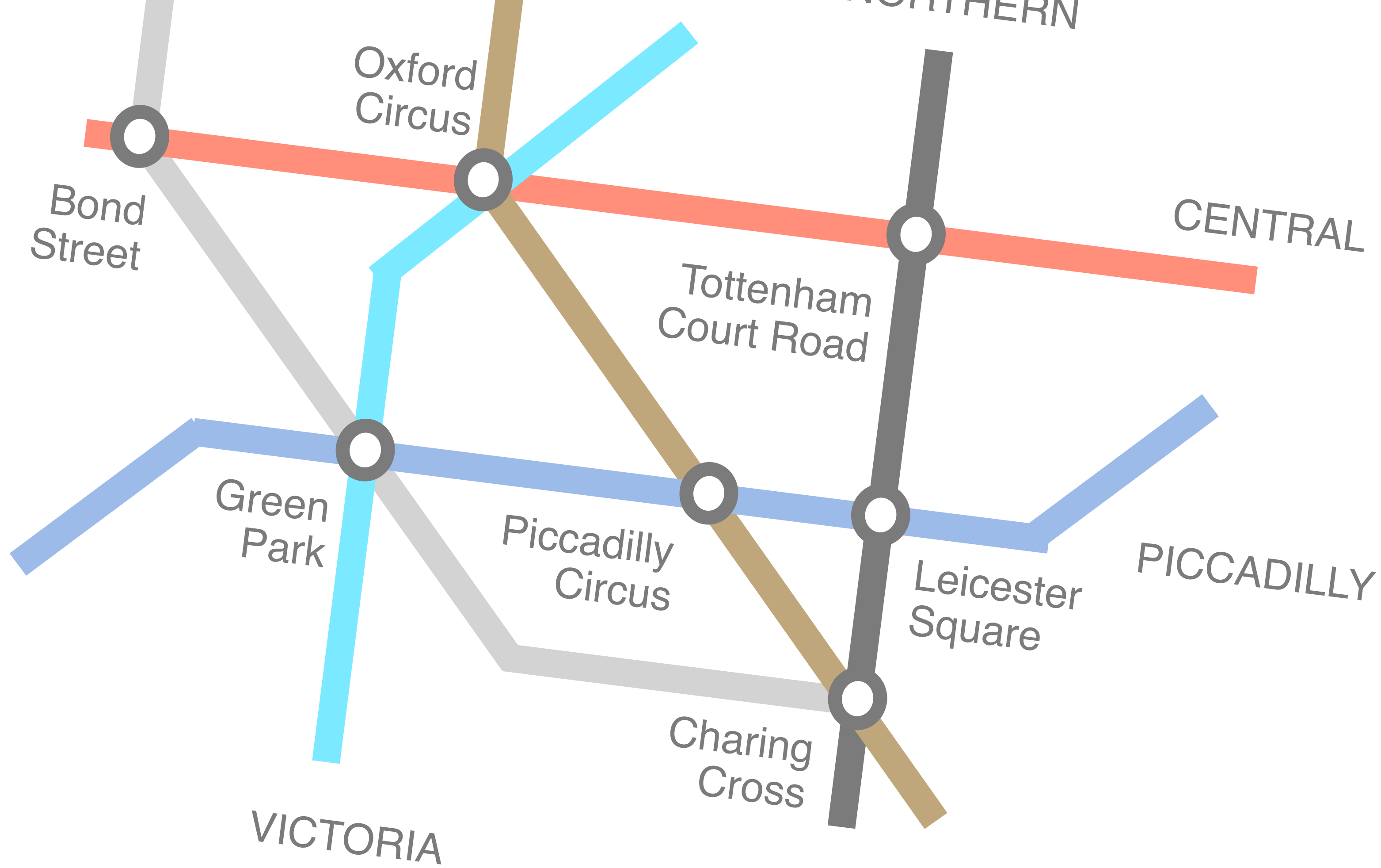
80-90

- 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

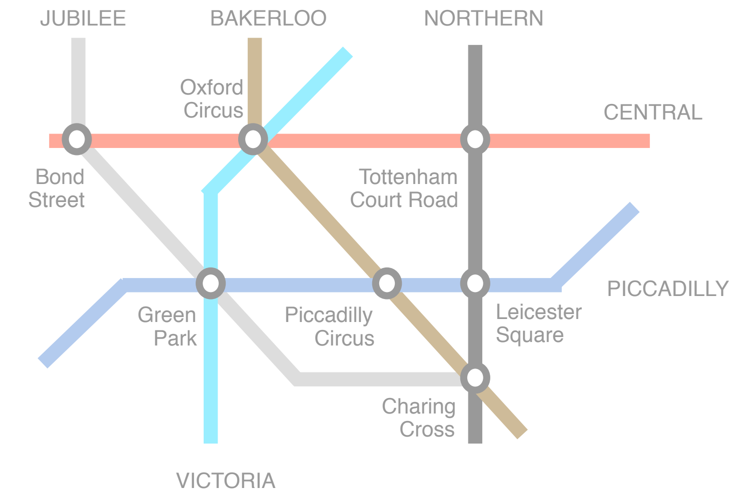
ternary connected/3:

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

binary nearby/2:

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

Representing Knowledge: *base information*

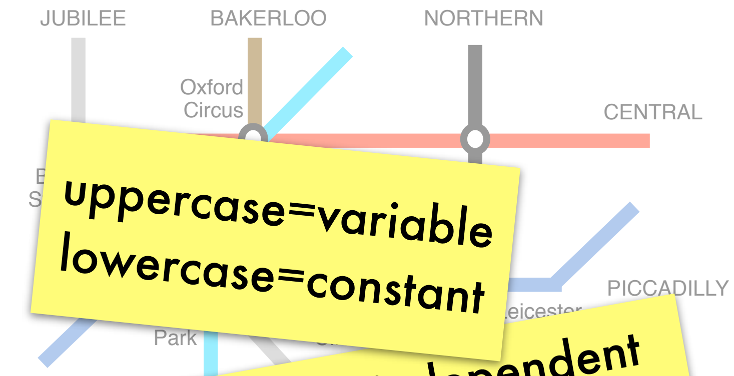


logic predicate connected/3 implemented through logic facts

```
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

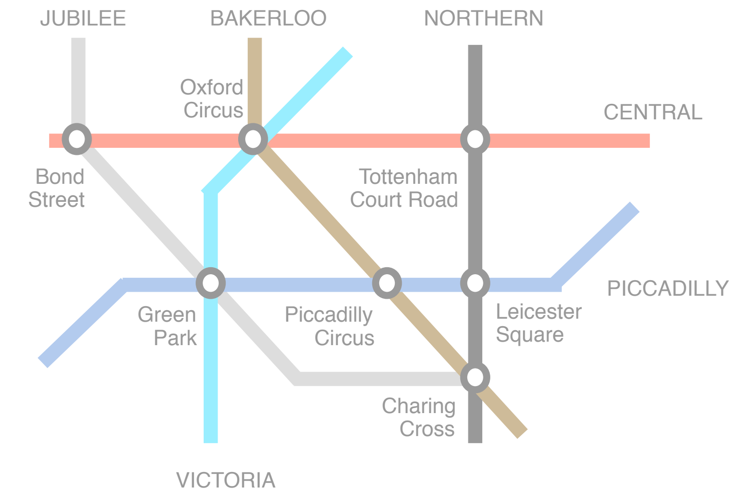
```
nearby(X,Y) :- connected(X,Z,L), connected(Z,Y,L).

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 }
```

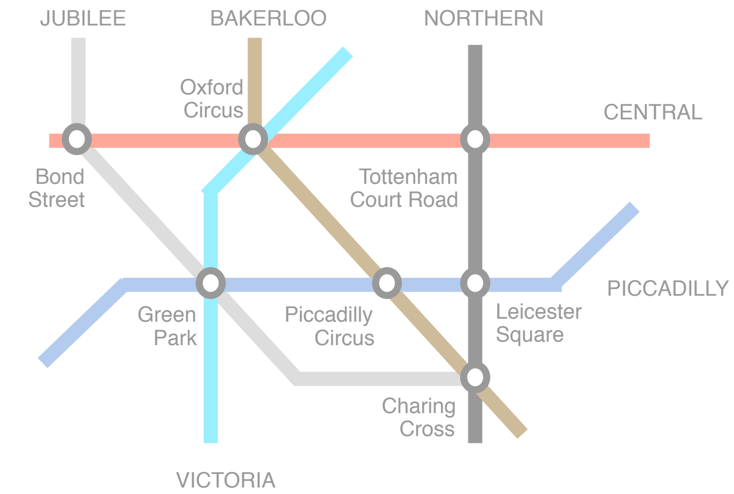
answer

```
{ 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:

premise of compatible rule

```
?- connected(tottenham_court_road, W, L).
```

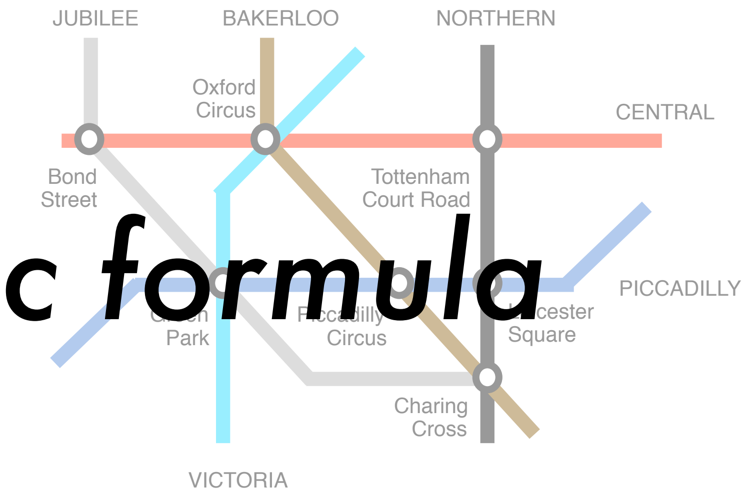
matching new predicate against a compatible logic fact yields:

```
{ W = leicester_square, L=northern}
```

final answer

```
{ X = tottenham_court_road, Y = leicester_square }
```

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



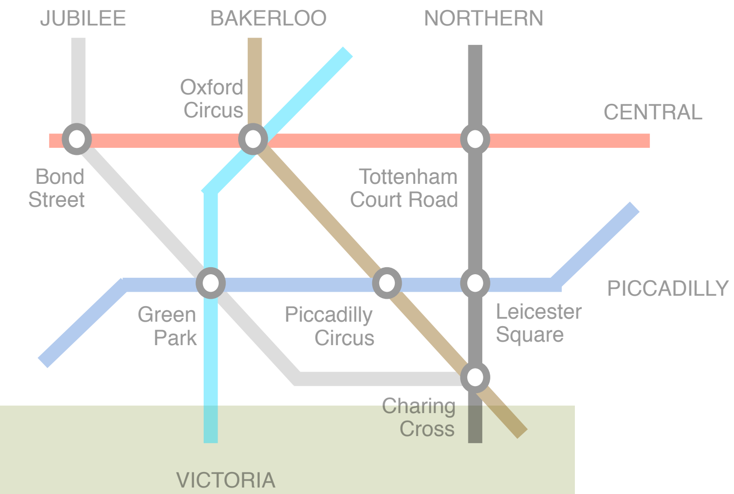

```
?- nearby(tottenham_court_road,W)
        |     nearby(X1,Y1) :- connected(X1,Y1,L1)
        |       { X1=tottenham_court_road, Y1=W }
        v
?- connected(tottenham_court_road,W,L1)
        |   connected(tottenham_court_road, leicester_square)
        |     { W=leicester_square,L1=northern}
 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

logic rule (with variables renamed for uniqueness)

logic fact

answer

Answering Queries: *involving recursive rules*



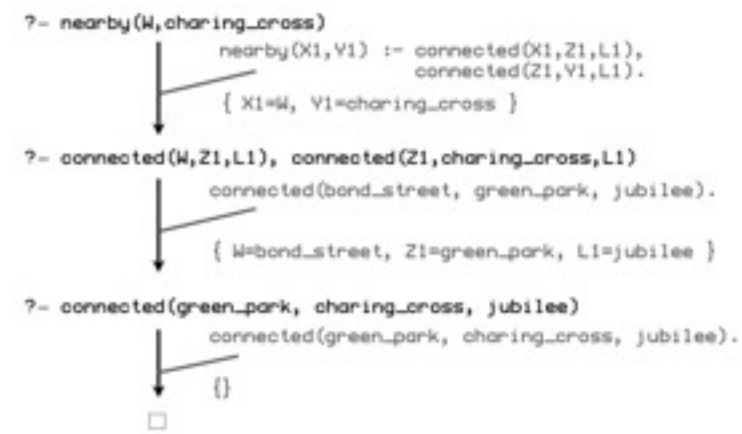

```
reachable(X,Y) :- connected(X,Y,L).
reachable(X,Y) :- connected(X,Z,L), reachable(Z,Y).
```

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

left-most condition expanded first

different rule applications different variables

Prolog's Proof Strategy: *resolution principle*



resolution principle

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

find a compatible rule `A :- B1, ..., Bm` such that `A` matches `Q1`

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)`

↓

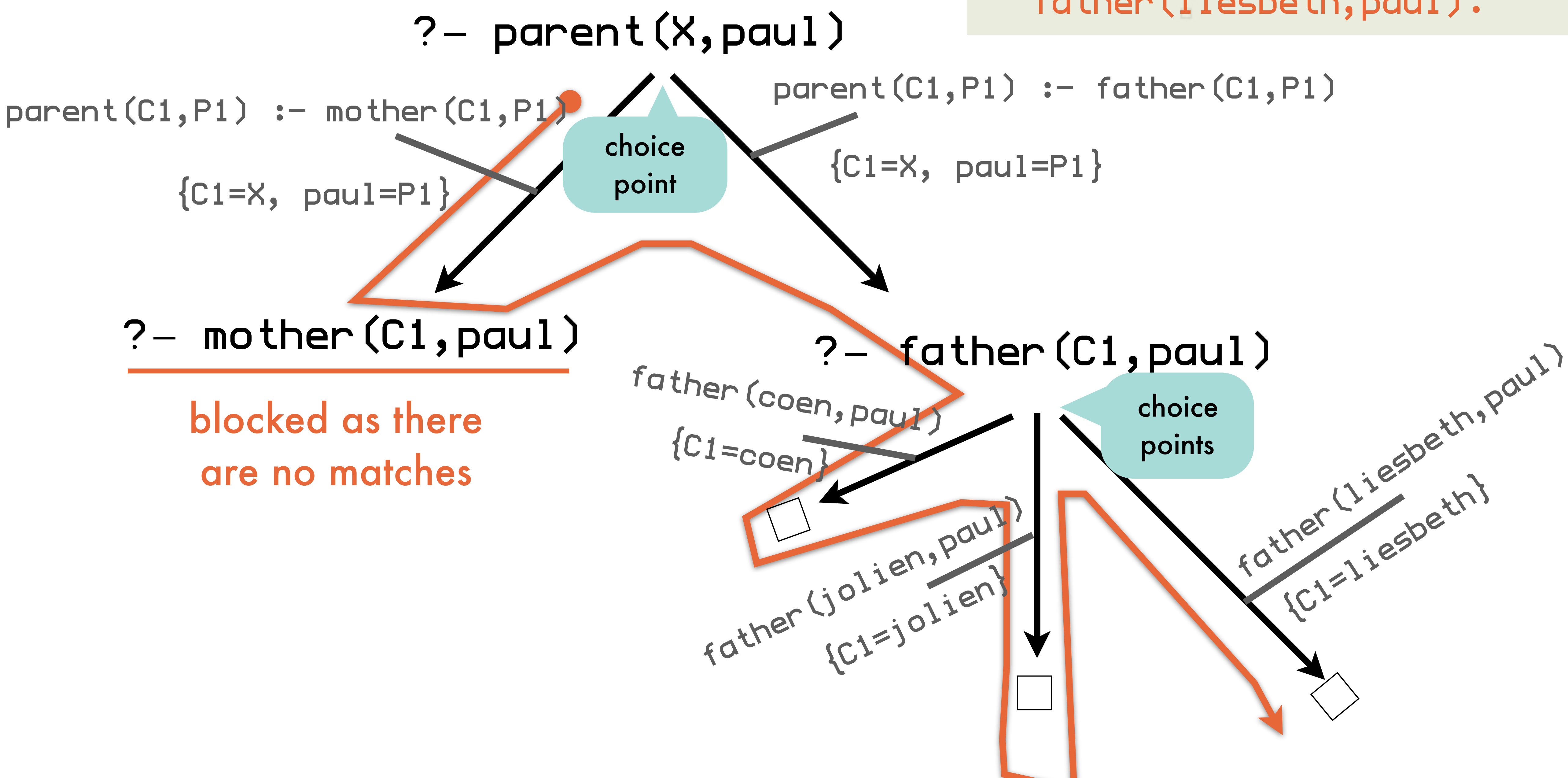
□ to a contradiction

“empty rule”:
premises are always true
conclusion is always false

in that case, the query is said “to succeed”

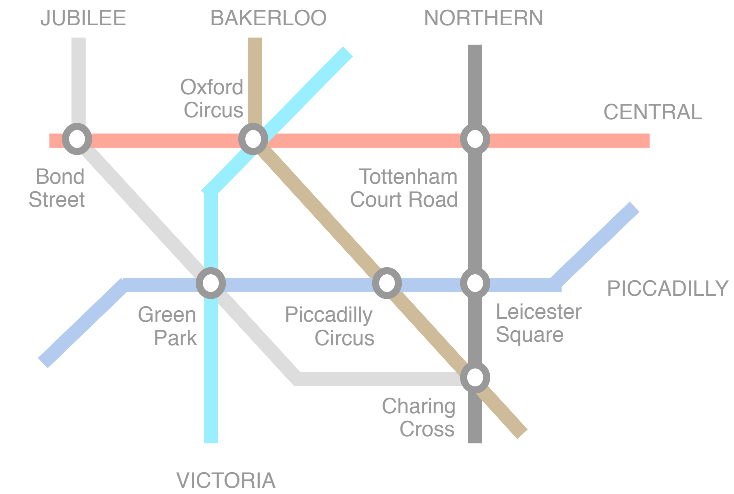
Prolog's Proof Strategy: *searching for a proof*

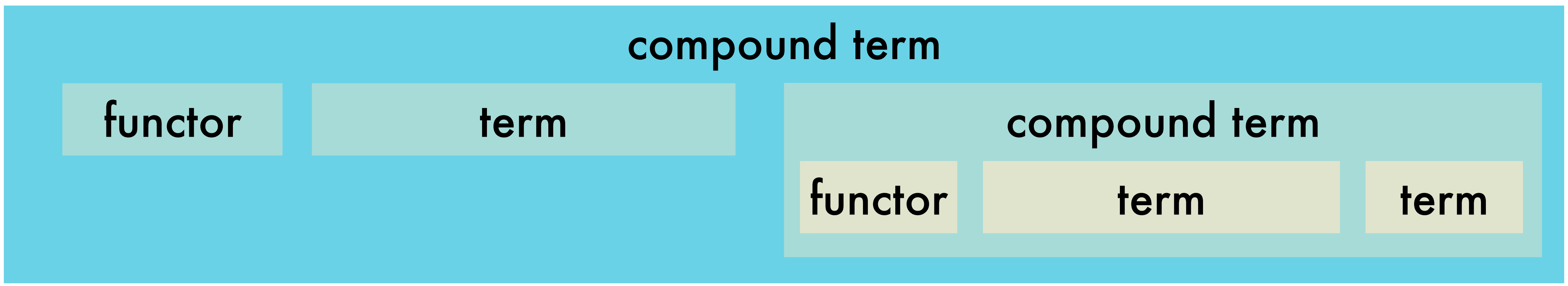
```
parent(C,P):- mother(C,P).
parent(C,P):- father(C,P).

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



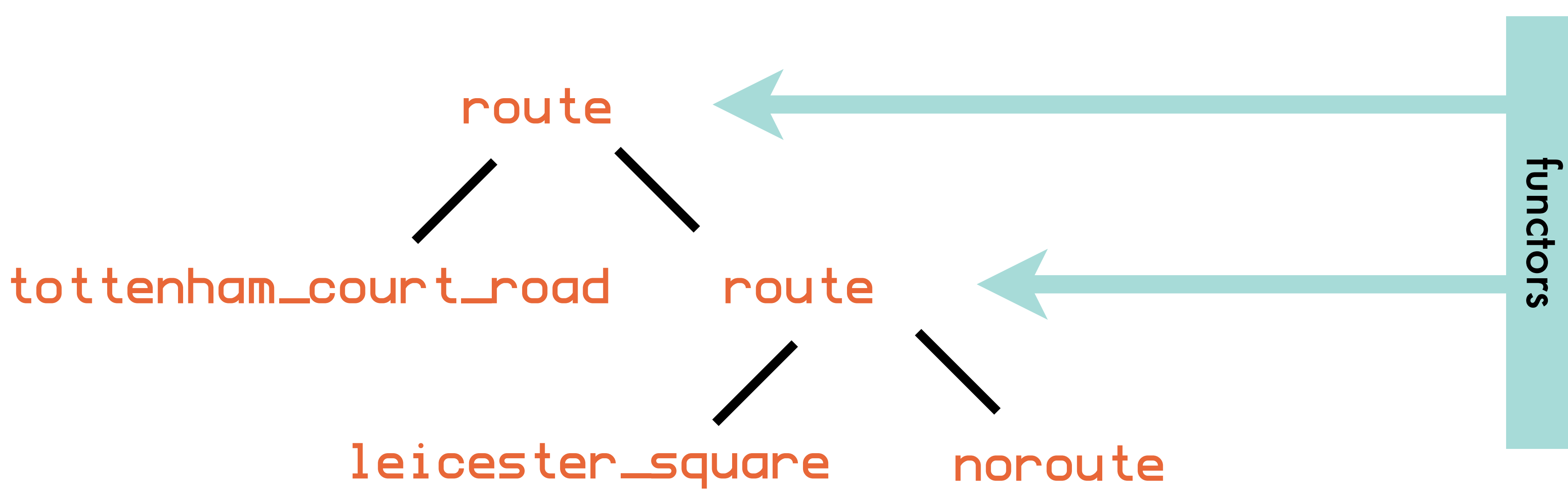
Prolog uses **depth-first search** to find a proof. When blocked or more answers are requested, it **backtracks** to the last choice point. Of multiple conditions, the **left-most** is tried first. Matching rules and facts are tried in the given order.

Representing Knowledge: *compound terms*

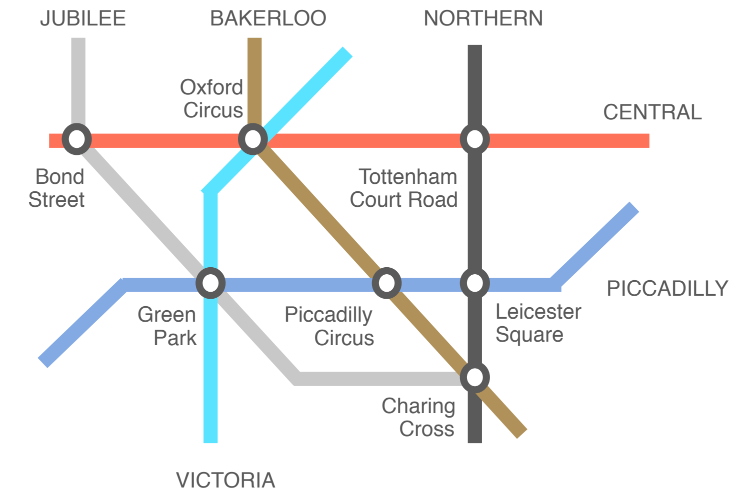





```
route(tottenham_court_road, route(leicester_square, noroute))
```



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!!

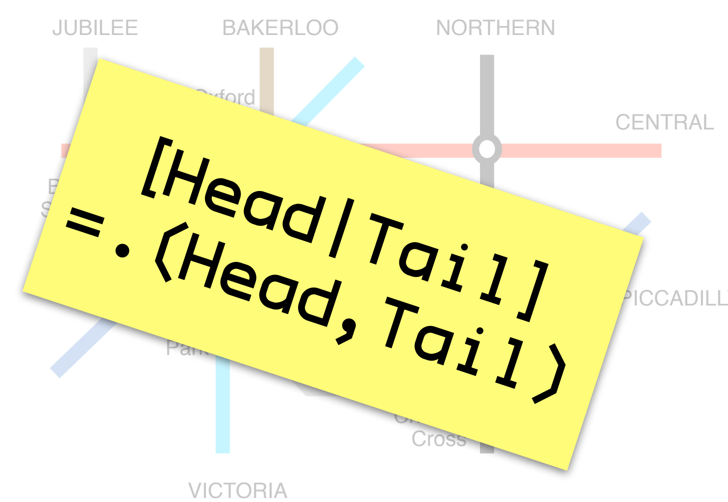
```
?- reachable(oxford_circus,charing_cross,R).
```

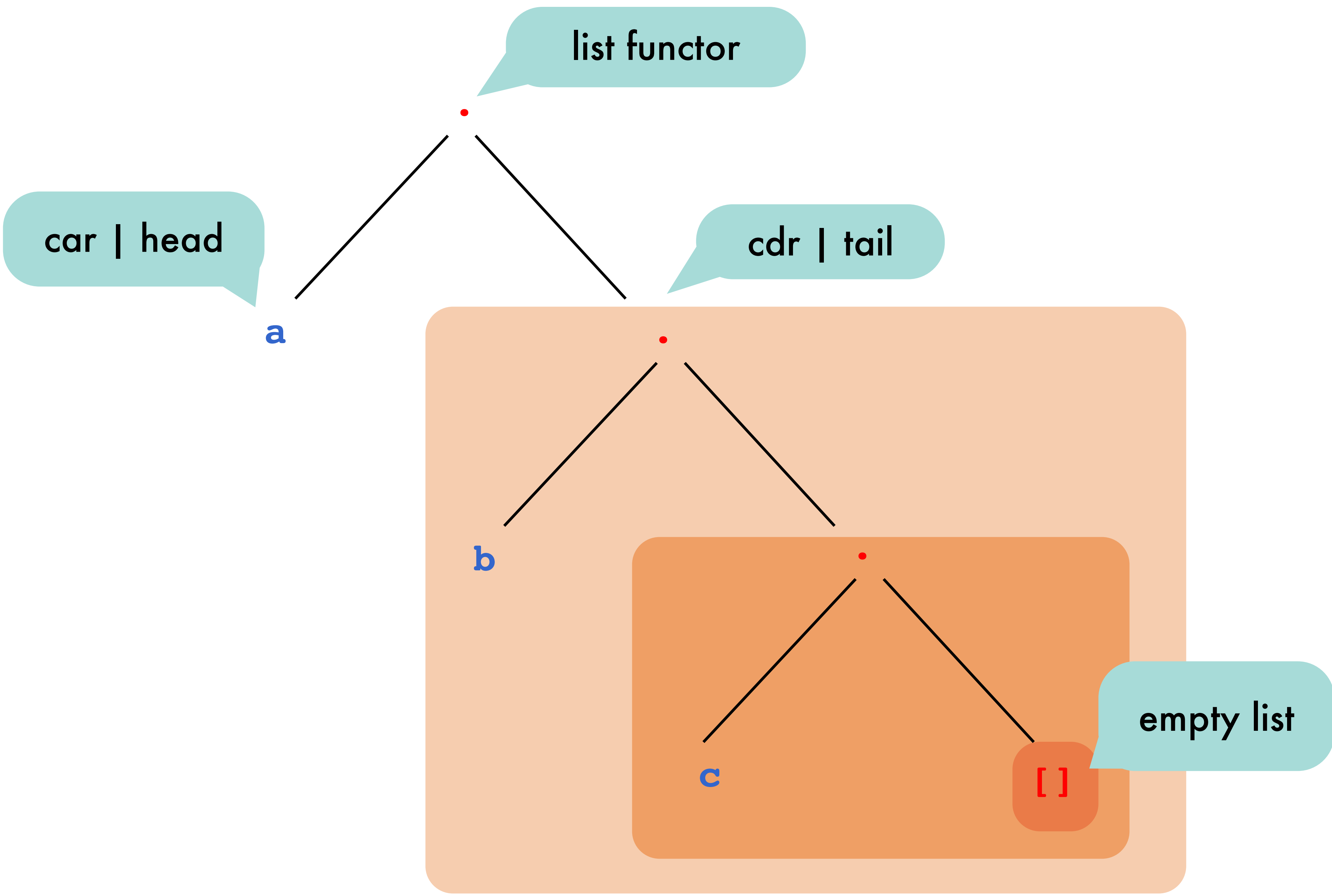
answer →
```
{ R = route(tottenham_court_road,
            route(leicester_square,noroute)) }
```

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]]]

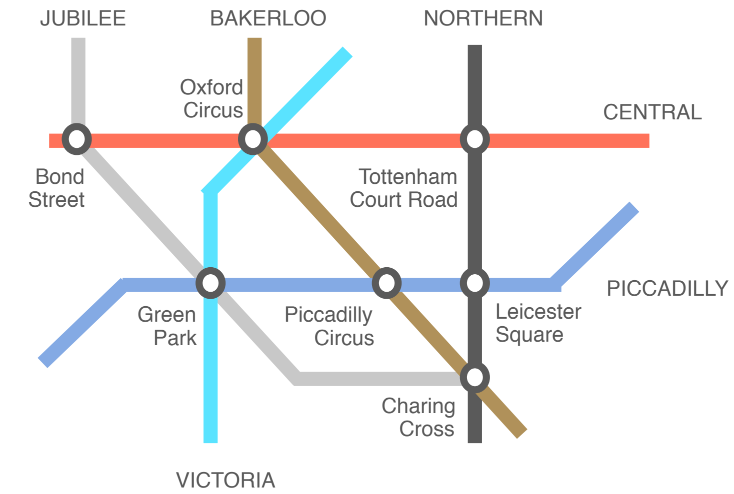
[a| [b,c]]

[a,b| [c]]

. . .

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

Representing Knowledge: *lists*



arbitrary length →

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

even length →

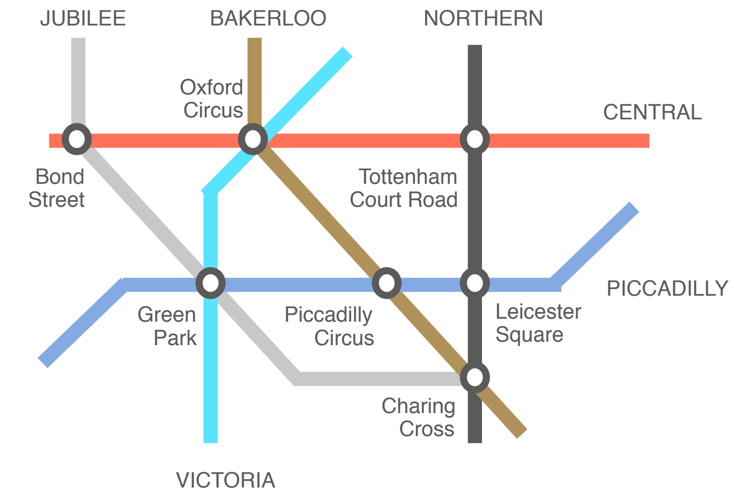
```
evenlist([]).
evenlist([First,Second|Rest]) :- evenlist(Rest).
```

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

possible because of the relational nature of logic programming

```
?- 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= []}`

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).
```

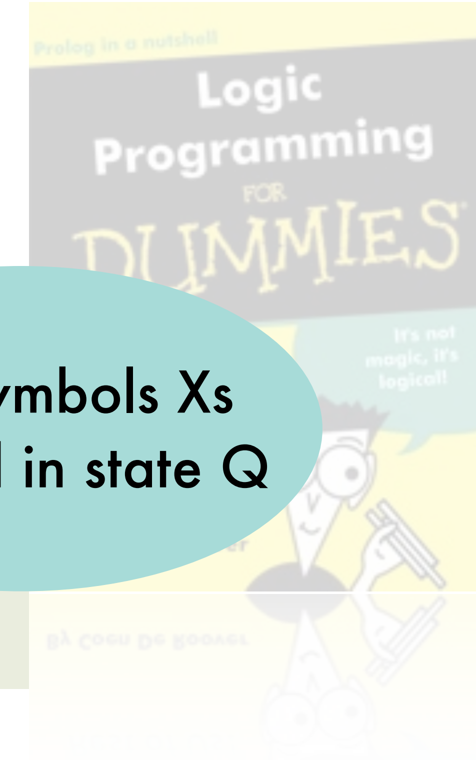
selection

```
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

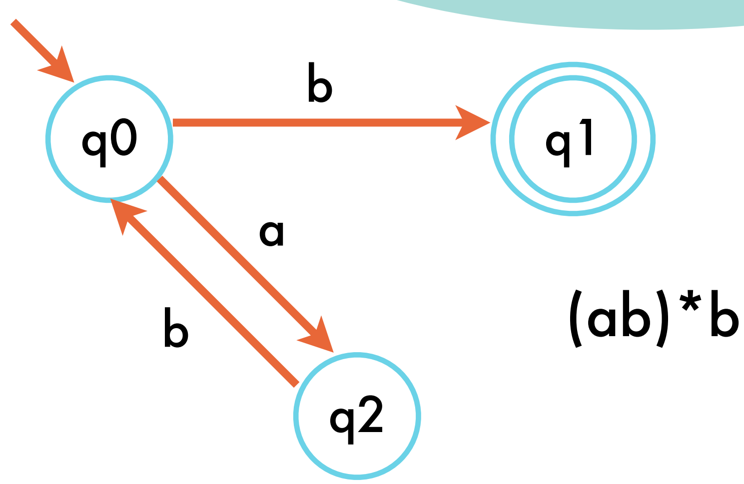



```
accept(Xs) :- initial(Q), accept(Xs,Q).

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




```
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*

decprog1_dfa.pl

```prolog
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|...]
```

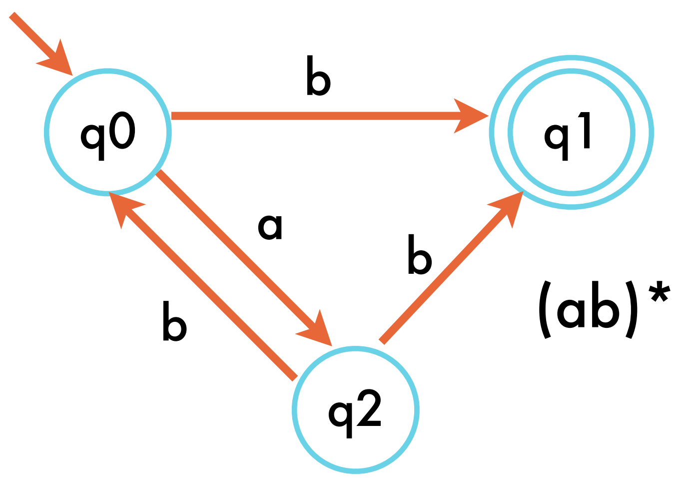
1:**- *prolog* 57% (15,0) (Inferior Prolog: run)

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]



(ab)*(ab|b)

```
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

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).
```

X pushed on stack

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

variable X substitutes for a concrete symbol !!

X popped off stack

[The Art of Prolog, Sterling&Shapiro]