CSc 372

Comparative Programming Languages

22 : Prolog — Introduction

Department of Computer Science
University of Arizona

collberg@gmail.com

What is Prolog?

What is Prolog?

- Prolog is a language which approaches problem-solving in a *declarative* manner. The idea is to define *what* the problem is, rather than *how* it should be solved.
- In practice, most Prolog programs have a procedural as well as a declarative component — the procedural aspects are often necessary in order to make the programs execute efficiently.

What is Prolog?

Algorithm = Logic + Control Robert A. Kowalski

Prescriptive Languages:

- Describe *how* to solve problem
- Pascal, C, Ada,...
- Also: Imperative, Procedural

Descriptive Languages:

- Describe *what* should be done
- Also: Declarative

Kowalski's equation says that

- Logic – is the specification (what the program should do)
- Control – what we need to do in order to make our logic execute efficiently. This usually includes imposing an execution order on the rules that make up our program.

Objects & Relationships

Objects & Relationships

Prolog programs deal with

- objects, and
- relationships between objects

English:

"Christian likes the record"

Prolog:

```
likes(christian, record).
```

Facts

Record Database

- Here's an excerpt from Christian's record database:

```
is_record(planet_waves).
is_record(desire).
is_record(slow_train).

recorded_by(planet_waves, bob_dylan).
recorded_by(desire, bob_dylan).
recorded_by(slow_train, bob_dylan).

recording_year(planet_waves, 1974).
recording_year(desire, 1975).
recording_year(slow_train, 1979).
```

Record Database...

- The data base contains *unary facts* (`is_record`) and *binary facts* (`recorded_by`, `recording_year`).
- The fact

  `is_record(slow_train)`

  can be interpreted as

  `slow_train` ***is-a-record***
- The fact `recording_year(slow_train, 1979)` can be interpreted as *the recording year of slow_train was 1979*.

Conditional Relationships

Conditional Relationships

- Prolog programs deal with conditional relationships between objects.

English:

"C. likes Bob Dylan records recorded before 1979"

Prolog:

```
likes(christian, X) :-
      is_record(X),
      recorded_by(X, bob_dylan),
      recording_year(X, Year),
      Year < 1979.
```

Conditional Relationships. . .

- The rule

  ```
  likes(christian, X) :-
        is_record(X),
        recorded_by(X, bob_dylan),
        recording_year(X, Year),
        Year < 1979.
  ```

  can be restated as

  > *"Christian likes* `X`*, if* `X` *is a record, and* `X` *is recorded by Bob Dylan, and the recording year is before 1979."*

- Variables start with capital letters.
- Comma (",") is read as *and*.

Asking Questions

Asking Questions

Prolog programs

- solve problems by asking questions.

English:

"Does Christian like the albums Planet Waves & Slow Train*?'*

Prolog:

```
?- likes(christian, planet_waves).
yes
?- likes(christian, slow_train).
no
```

Asking Questions. . .

English:

"Was Planet Waves *recorded by Bob Dylan?"*
"When was Planet Waves *recorded?"*
"Which album was recorded in 1974?"

Prolog:

```
?- recorded_by(planet_waves, bob_dylan).
   yes

?- recording_year(planet_waves, X).
   X = 1974

?- recording_year(X, 1974).
   X = planet_waves
```

Asking Questions. . .

In Prolog

- "," (a comma), means "and'

English:

"Did Bob Dylan record an album in 1974?"

Prolog:

```
?- is_record(X),
      recorded_by(X, bob_dylan),
      recording_year(X, 1974).
yes
```

Asking Questions...

Sometimes a query has more than one answer:

- Use `";"` to get all answers.

English:

"What does Christian like?"

Prolog:

```
?- likes(christian, X).
   X = planet_waves ;

   X = desire ;

no
```

Asking Questions…

Sometimes answers have more than one part:

English:

"List the albums and their artists!"

Prolog:

```
?- is_record(X), recorded_by(X, Y).
X = planet_waves,
Y = bob_dylan ;
X = desire,
Y = bob_dylan ;
X = slow_train,
Y = bob_dylan ;
no
```

Recursive Rules

Recursive Rules

> *"People are influenced by the music they listen to.*
> *People are influenced by the music listened to by the people* they *listen to."*

```
listens_to(bob_dylan, woody_guthrie).
listens_to(arlo_guthrie, woody_guthrie).
listens_to(van_morrison, bob_dylan).
listens_to(dire_straits, bob_dylan).
listens_to(bruce_springsteen, bob_dylan).
listens_to(björk, bruce_springsteen).

influenced_by(X, Y) :- listens_to(X, Y).
influenced_by(X, Y) :- listens_to(X,Z),
                       influenced_by(Z,Y).
```

Asking Questions. . .

English:

"Is Björk influenced by Bob Dylan?"
"Is Björk influenced by Woody Guthrie?"
"Is Bob Dylan influenced by Bruce Springsteen?"

Prolog:

```
?- influenced_by(bjork, bob_dylan).
yes
?- influenced_by(bjork, woody_guthrie).
yes
?- influenced_by(bob_dylan, bruce_s).
no
```

Visualizing Logic

- *Comma* (`,`) is read as `and` in Prolog. Example: The rule

  ```
  person(X) :- has_bellybutton(X), not_dead(X).
  ```

  is read as

  > *"X is a person if X has a bellybutton and X is not dead."*

- *Semicolon* (`;`) is read as `or` in Prolog. The rule

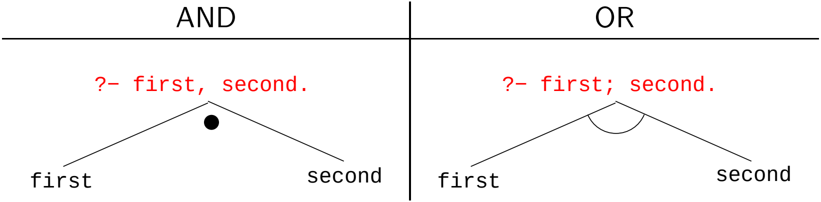
  ```
  person(X) :- X=adam ; X=eve ;
                     has_bellybutton(X).
  ```

  is read as

  > *"X is a person if X is adam or X is eve or X has a bellybutton."*

Visualizing Logic…

- To visualize what happens when Prolog executes (and this can often be very complicated!) we use the following two notations:

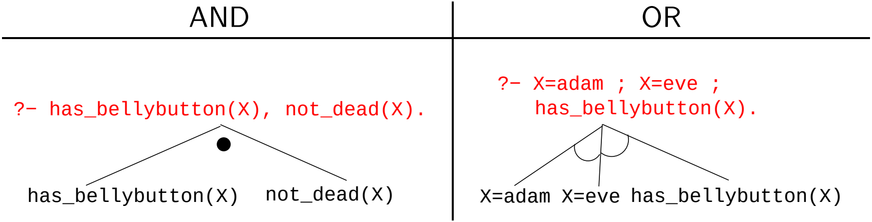


AND	OR
`?- first, second.`	`?- first; second.`
`first` `second`	`first` `second`

- For `AND`, both legs have to succeed.
- For `OR`, one of the legs has to succeed.

Visualizing Logic. . .

- Here are two examples:



AND

```
?- has_bellybutton(X), not_dead(X).
```

`has_bellybutton(X)` `not_dead(X)`

OR

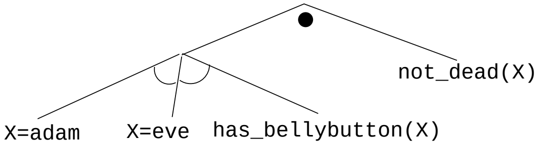
```
?- X=adam ; X=eve ;
   has_bellybutton(X).
```

`X=adam` `X=eve` `has_bellybutton(X)`

Visualizing Logic. . .

- `and` and `or` can be combined:

  `?- (X=adam ; X=eve ; has_bellybutton(X)), not_dead(X).`



- This query asks

  "Is there a person X who is adam, eve, or who has a bellybutton, and who is also not dead?"

How does Prolog Answer Questions?

Answering Questions

```
(1)    scientist(helder).
(2)    scientist(ron).
(3)    portuguese(helder).
(4)    american(ron).
(5)    logician(X) :- scientist(X).
(6)    ?- logician(X), american(X).
```

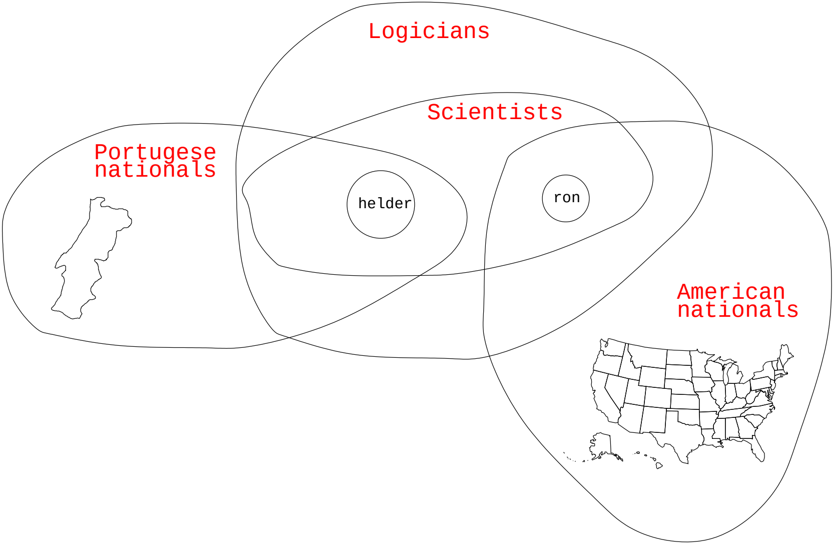
- The rule (5) states that

  "Every scientist is a logician"

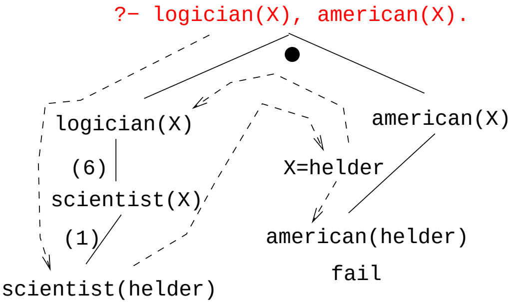
- The question (6) asks

  "Which scientist is a logician and *an american?"*

Answering Questions. . .

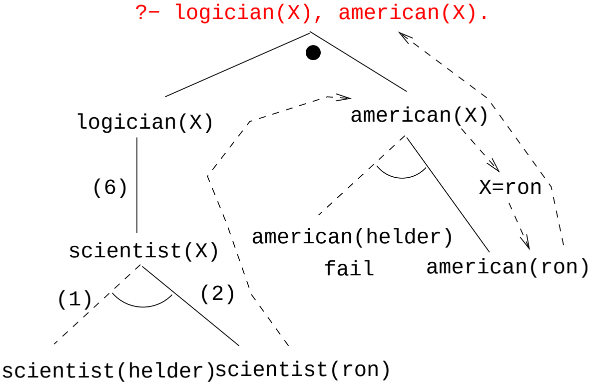


Answering Questions. . .



```
(1)   scientist(helder).
(2)   scientist(ron).
(3)   portuguese(helder).
(4)   american(ron).
(5)   logician(X) :- scientist(X).
(6)   ?- logician(X), american(X).
```

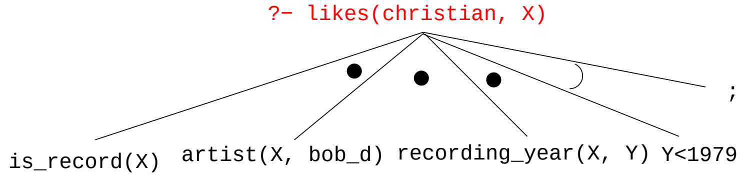
Answering Questions. . .



Answering Questions. . .

```
is_record(planet_waves).  is_record(desire).
is_record(slow_train).

recorded_by(planet_waves, bob_dylan).
recorded_by(desire, bob_dylan).
recorded_by(slow_train, bob_dylan).

recording_year(planet_waves, 1974).
recording_year(desire, 1975).
recording_year(slow_train, 1979).

likes(christian, X) :-
      is_record(X), recorded_by(X, bob_dylan),
      recording_year(X, Year), Year < 1979.
```

Answering Questions. . .



?- likes(christian, X)

is_record(X) artist(X, bob_d) recording_year(X, Y) Y<1979 ;

X = planet_waves Y=1979 succeed

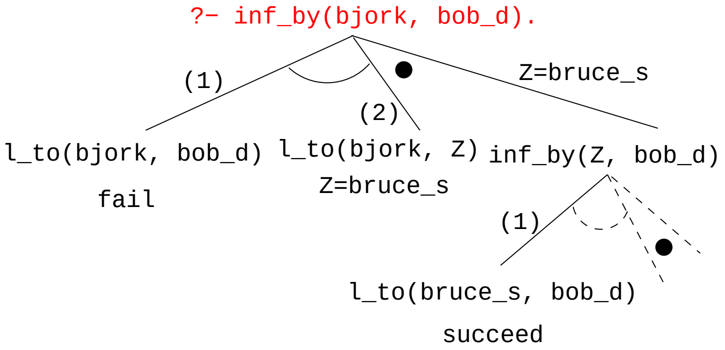
X = desire Y=1975 succeed

X = slow_train Y=1974 fail

Answering Questions...

```
listens_to(bob_dylan, woody_guthrie).
listens_to(arlo_guthrie, woody_guthrie).
listens_to(van_morrison, bob_dylan).
listens_to(dire_straits, bob_dylan).
listens_to(bruce_springsteen, bob_dylan).
listens_to(björk, bruce_springsteen).

(1)   influenced_by(X, Y) :- listens_to(X, Y).
(2)   influenced_by(X, Y) :-
          listens_to(X, Z),
          influenced_by(Z, Y).

?- influenced_by(bjork, bob_dylan).
?- inf_by(bjork, woody_guthrie).
```

Answering Questions. . .



?- inf_by(bjork, bob_d).

(1)

(2)

●

Z=bruce_s

l_to(bjork, bob_d) l_to(bjork, Z) inf_by(Z, bob_d)

fail

Z=bruce_s

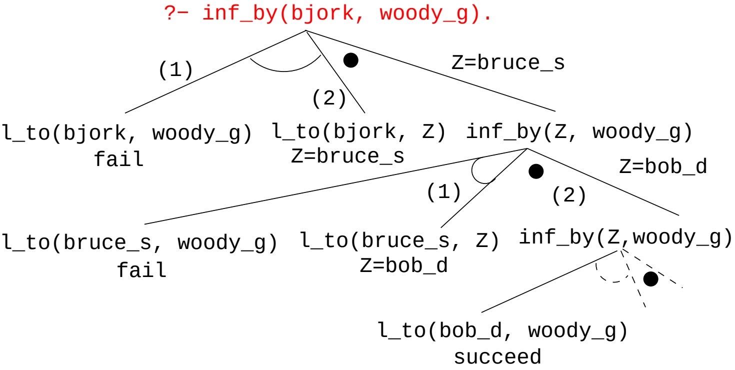
(1)

●

l_to(bruce_s, bob_d)

succeed

Answering Questions. . .



?- inf_by(bjork, woody_g).

(1)

(2)

Z=bruce_s

l_to(bjork, woody_g)
fail

l_to(bjork, Z)
Z=bruce_s

inf_by(Z, woody_g)

(1)

(2)

Z=bob_d

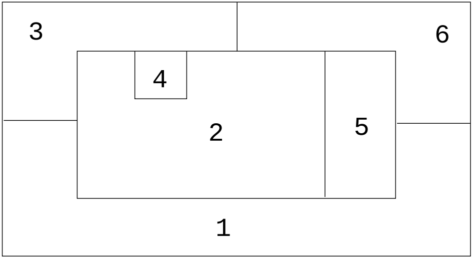
l_to(bruce_s, woody_g)
fail

l_to(bruce_s, Z)
Z=bob_d

inf_by(Z,woody_g)

l_to(bob_d, woody_g)
succeed

Map Coloring



"Color a planar map with at most four colors, so that contiguous regions are colored differently."

Map Coloring. . .

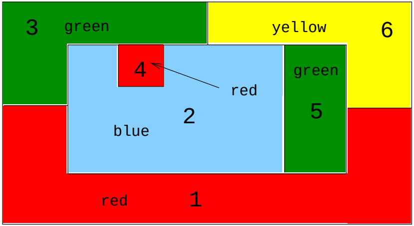
A coloring is OK iff

1. The color of Region 1 $\neq$ the color of Region 2, and
2. The color of Region 1 $\neq$ the color of Region 3,...

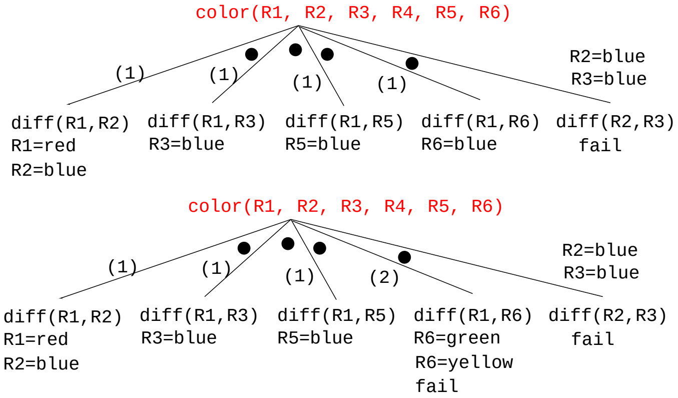
```
color(R1, R2, R3, R4, R5, R6) :-
   diff(R1, R2), diff(R1, R3), diff(R1, R5), diff(R1, R6),
   diff(R2, R3), diff(R2, R4), diff(R2, R5), diff(R2, R6),
   diff(R3, R4), diff(R3, R6), diff(R5, R6).

diff(red,blue).  diff(red,green).  diff(red,yellow).
diff(blue,red).  diff(blue,green).  diff(blue,yellow).
diff(green,red).  diff(green,blue).  diff(green,yellow).
diff(yellow, red).diff(yellow,blue).  diff(yellow,green).
```

Map Coloring...

```
?- color(R1, R2, R3, R4, R5, R6).
R1 = R4 = red, R2 = blue,
R3 = R5 = green, R6 = yellow ;

R1 = red, R2 = blue,
R3 = R5 = green, R4 = R6 = yellow
```



Map Coloring – Backtracking



color(R1, R2, R3, R4, R5, R6)

(1) (1) (1) (1)

R2=blue
R3=blue

diff(R1,R2)
R1=red
R2=blue

diff(R1,R3)
R3=blue

diff(R1,R5)
R5=blue

diff(R1,R6)
R6=blue

diff(R2,R3)
fail

color(R1, R2, R3, R4, R5, R6)

(1) (1) (1) (2)

R2=blue
R3=blue

diff(R1,R2)
R1=red
R2=blue

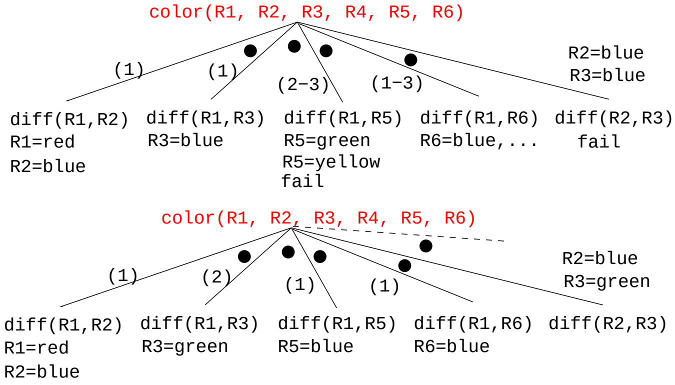
diff(R1,R3)
R3=blue

diff(R1,R5)
R5=blue

diff(R1,R6)
R6=green
R6=yellow
fail

diff(R2,R3)
fail

Map Coloring – Backtracking



color(R1, R2, R3, R4, R5, R6)

(1) (1) (2–3) (1–3)

R2=blue
R3=blue

diff(R1,R2)
R1=red
R2=blue

diff(R1,R3)
R3=blue

diff(R1,R5)
R5=green
R5=yellow
fail

diff(R1,R6)
R6=blue,...

diff(R2,R3)
fail

color(R1, R2, R3, R4, R5, R6)

(1) (2) (1) (1)

R2=blue
R3=green

diff(R1,R2)
R1=red
R2=blue

diff(R1,R3)
R3=green

diff(R1,R5)
R5=blue

diff(R1,R6)
R6=blue

diff(R2,R3)

Working with gprolog

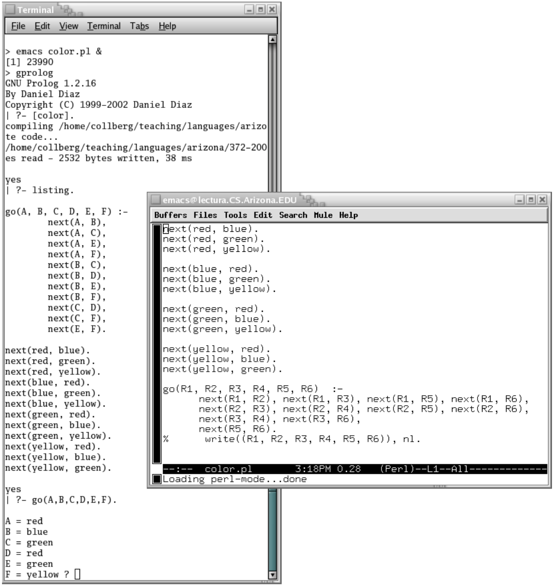
- `gprolog` can be downloaded from here: http://gprolog.inria.fr/.
- `gprolog` is installed on `lectura` (it's also on the Windows machines) and is invoked like this:

```
> gprolog
GNU Prolog 1.2.16
| ?- [color].
| ?- listing.
go(A, B, C, D, E, F) :- next(A, B), ...
| ?- go(A,B,C,D,E,F).
A = red ...
```

Working with `gprolog`...

- The command `[color]` loads the prolog program in the file `color.pl`.
- You should use the texteditor of your choice (`emacs`, `vi`,...) to write your prolog code.
- The command `listing` lists all the prolog predicates you have loaded.

Working with `gprolog`...



```
> emacs color.pl &
[1] 23990
> gprolog
GNU Prolog 1.2.16
By Daniel Diaz
Copyright (C) 1999-2002 Daniel Diaz
| ?- [color].
compiling /home/collberg/teaching/languages/arizo
te code...
/home/collberg/teaching/languages/arizona/372-200
es read - 2532 bytes written, 38 ms

yes
| ?- listing.

go(A, B, C, D, E, F) :-
        next(A, B),
        next(A, C),
        next(A, E),
        next(A, F),
        next(B, C),
        next(B, D),
        next(B, E),
        next(B, F),
        next(C, D),
        next(C, F),
        next(E, F).

next(red, blue).
next(red, green).
next(red, yellow).
next(blue, red).
next(blue, green).
next(blue, yellow).
next(green, red).
next(green, blue).
next(green, yellow).
next(yellow, red).
next(yellow, blue).
next(yellow, green).

yes
| ?- go(A,B,C,D,E,F).

A = red
B = blue
C = green
D = red
E = green
F = yellow ?
```

```
next(red, blue).
next(red, green).
next(red, yellow).

next(blue, red).
next(blue, green).
next(blue, yellow).

next(green, red).
next(green, blue).
next(green, yellow).

next(yellow, red).
next(yellow, blue).
next(yellow, green).

go(R1, R2, R3, R4, R5, R6)  :-
       next(R1, R2), next(R1, R3), next(R1, R5), next(R1, R6),
       next(R2, R3), next(R2, R4), next(R2, R5), next(R2, R6),
       next(R3, R4), next(R3, R6),
       next(R5, R6).
%       write((R1, R2, R3, R4, R5, R6)), nl.
--:--  color.pl         3:18PM 0.28    (Perl)--L1--All-------------
Loading perl-mode...done
```

Readings and References

- Read Clocksin-Mellish, Chapter 1-2.
- `http://dmoz.org/Computers/Programming/Languages/Prolog`

Prolog by Example	Coelho & Cotta
Prolog: Programming for AI	Bratko
Programming in Prolog	Clocksin & Mellish
The Craft of Prolog	O'Keefe
Prolog for Programmers	Kluzniak & Szpakowicz
Prolog	Alan G. Hamilton
The Art of Prolog	Sterling & Shapiro

Readings and References. . .

Computing with Logic	Maier & Warren
Knowledge Systems Through Prolog	Steven H. Kim
Natural Language Processing in Prolog	Gazdar & Mellish
Language as a Cognitive Process	Winograd
Prolog and Natural Language Analysis	Pereira and Shieber
Computers and Human Language	George W. Smith
Introduction to Logic	Irving M. Copi
Beginning Logic	E.J.Lemmon

Prolog So Far

- A Prolog program consists of a number of *clauses*:

Rules

- Have `head` + `body`:

$$\overbrace{\texttt{likes(chris, X)}}^{\text{head}}\ \texttt{:-}$$

$$\underbrace{\texttt{girl(X), black\_hair(X)}}_{\text{body}}$$

- Can be recursive

Facts

- Head but no body.
- Always true.

Prolog So Far. . .

- A clause consists of

atoms	Start with lower-case letter.
variables	Start with upper-case letter.

- Prolog programs have a
  - Declarative meaning
    - The relations defined by the program
  - Procedural meaning
    - The order in which goals are tried

Prolog So Far…

- A question consists of one or more goals:
  - `?- likes(chris, X), smart(X).`
  - `","` means `and`
  - Use `";"` to get all answers
  - Questions are either
    - Satisfiable (the goal succeeds)
    - Unsatisfiable (the goal fails)
  - Prolog answers questions (satisfies goals) by:
    - instantiating variables
    - searching the database sequentially
    - backtracking when a goal fails

CSc 372

Comparative Programming Languages

23 : Prolog — Basics

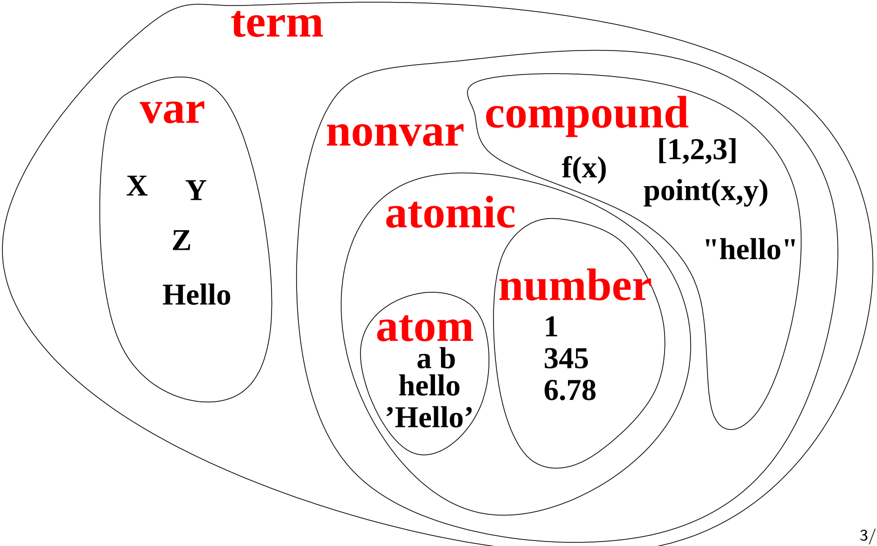
Department of Computer Science
University of Arizona

collberg@gmail.com

Prolog Types

- The term is Prolog's basic data structure.
- Everything is expressed in the form of a term. This includes programs and data.
- Prolog has four basic types of terms:
  1. variables start with an uppercase letter;
  2. compound terms are lists, strings, and structures;
  3. atoms start with a lower-case letter;
  4. numbers.

Prolog Types. . .



Prolog Numbers

- Most Prolog implementations support infinite precision integers. This is not true of GNU Prolog!
- The built-in operator is evaluates arithmetic expressions:

```
| ?- X is 6*7.
X = 42
| ?- X is 6.0*7.0.
X = 42.0
| ?- X is 60000000000000*7000000000000000.
X = 1
```

Prolog Arithmetic Expressions

- An infix expression is just shorthand for a structure:

```
| ?- X = +(1,*(2,3)).
X = 1+2*3
| ?- X = 1+2*3.
X = 1+2*3
| ?- X is +(1,*(2,3)).
X = 7
| ?- X is 1+2*3.
X = 7
```

- $X = 1*2$ means “make the variable X and 1*2 the same”. It looks like an assignment, but it's what we call unification. More about that later.

Prolog Atoms

- Atoms are similar to `enums` in C.
- Atoms start with a lower-case letter and can contain letters, digits, and underscore (_).

```
| ?- X = hello.
X = hello
| ?- X = hE_l_l_o99.
X = hE_l_l_o99
```

Prolog Variables

- Variables start out uninstantiated, i.e. without a value.
- Uninstantiated variables are written `_number`:

```
| ?- write(X).
_16
```

- Once a Prolog variable has been instantiated (given a value), it will keep that value.

```
| ?- X=sally.
X = sally
| ?- X=sally, X=lisa.
no
```

Prolog Variables...

- When a program backtracks over a variable instantiation, the variable again becomes uninstantiated.

```
| ?- (X=sally; X=lisa), write(X), nl.
sally
X = sally ?  ;

lisa
X = lisa
```

Prolog Programs

- A Prolog program consists of a database of facts and rules:

```
likes(lisa,chocolate).
likes(lisa,X) :- tastes_like_chocolate(X).
```

- `:-` is read if.
- `:-` is just an operator, like other Prolog operators. The following are equivalent:

```
likes(lisa,X) :- boy(X),tastes_like_choc(X).

:-(likes(lisa,X),
   (boy(X),tastes_like_chok(X))).
```

Prolog Programs. . .

- Prolog facts/rules can be overloaded, wrt their arity.
- You can have a both a rule `foo()` and a rule `foo(X)`:

```
| ?- [user].
foo.
foo(hello).
foo(bar,world).
foo(X,Y,Z) :-
    Z is X + Y.
<ctrl-D>
```

```
| ?- foo.
yes
| ?- foo(X).
X = hello
| ?- foo(X,Y).
X = bar
Y = world
| ?- foo(1,2,Z).
Z = 3
```

Standard predicates

- `read(X)` and `write(X)` read and write Prolog terms.
- `nl` prints a newline character.

```
| ?- write(hello),nl.
hello

| ?- read(X), write(X), nl.
hello.
hello
```

Standard predicates…

- `write` can write arbitrary Prolog terms:

```
| ?- write(hello(world)),nl.
hello(world)
```

- Note that `read(X)` requires the input to be syntactically correct and to end with a period.

```
| ?- read(X).
foo).
uncaught exception:  error
```

Unification/Matching

- The = -operator tries to make its left and right-hand sides the same.
- This is called unification or matching.
- If Prolog can't make `X` and `Y` the same in `X = Y`, matching will fail.

```
| ?- X=lisa, Y=sally, X = Y.
no
| ?- X=lisa, Y=lisa, Z = X, Z = Y.
X = lisa
Y = lisa
Z = lisa
```

- We will talk about this much more later.

Backtracking

- Prolog will try every possible way to satisfy a query.
- Prolog explores the search space by using backtracking, which means undoing previous computations, and exploring a different search path.

Backtracking...

- Here's an example:

```
| ?- [user].
girl(sally).
girl(lisa).
pretty(lisa).
blonde(sally).
| ?- girl(X),pretty(X).
X = lisa
| ?- girl(X),pretty(X),blonde(X).
no
| ?- (X=lisa; X=sally), pretty(X).
X = lisa
```

- We will talk about this much more later.

Māori Family Relationships

John Foster (in *He Whakamaarama – A New Course in Māori*) writes:

> *Relationship is very important to the Māori. Social seniority is claimed by those able to trace their whakapapa or genealogy in the most direct way to illustrious ancestors. Rights to shares in land and entitlement to speak on the marae may also depend on relationship. Because of this, there are special words to indicate elder or younger relations, or senior or younger branches of a family.*

- Māori is the indigenous language spoken in New Zealand. It is a polynesian language, and closely related to the language spoken in Hawaii.

Māori Terms of Address

Māori	English
au	I
tipuna, tupuna	grandfather, grandmother, grandparent, ancestor
tiipuna	grandparents
matua taane	father
maatua	parents
paapaa	father
whaea, maamaa	mother
whaea kee	aunt
kuia	grandmother, old lady
tuakana	older brother of a man, older sister of a woman
teina	younger brother of a man, younger sister of a woman

Māori Terms of Address. . .

Māori	English
tungaane	woman's brother (older or younger)
tuahine	man's sister (older or younger)
kaumaatua	elder (male)
mokopuna	grandchild (male or female)
iraamutu	niece, nephew
taane	husband, man
hunaonga	daughter-in-law, son-in-law
tamaahine	daughter
tama	son
tamaiti	child (male or female)
tamariki	children
wahine	wife, woman
maataamua	oldest child

Māori Terms of Address. . .

Māori	English
pootiki	youngest child
koroheke, koro, koroua	old man
whaiapo	boyfriend, girlfriend[1]
kootiro	girl
tamaiti taane	boy
whanaunga	relatives

---

[1] Literally: "What you follow at night"

The Whanau

- A program to translate between English and Māori must take into account the differences in terms of address between the two languages.
- Write a Prolog predicate `calls(X,Y,Z)` which, given a database of family relationships, returns **all** the words that `X` can use to address or talk about `Y`.

```
?- calls(aanaru, hata, Z).
   Z = tuakana ;
   Z = maataamua ;
   no

?- calls(aanaru, rapeta, Z).
   Z = teina ;
   no
```

The Whanau...

- Whanau is Māori for family.
- Below is a table showing an extended Māori family.

Name	Sex	Father	Mother	Spouse	Born
Hoone	male	unknown	unknown	Rita	1910
Rita	female	unknown	unknown	Hone	1915
Ranginui	male	unknown	unknown	Reremoana	1915
Reremoana	female	unknown	unknown	Ranginui	1916
Rewi	male	Hoone	Rita	Rahia	1935
Rahia	female	Ranginui	Reremoana	Rewi	1940
Hata	male	Rewi	Rahia	none	1957
Kiri	female	Rewi	Rahia	none	1959

The Whanau. . .

Name	Sex	Father	Mother	Spouse	Born
Hiniera	female	Rewi	Rahia	Pita	1960
Aanaru	male	Rewi	Rahia	none	1962
Rapeta	male	Rewi	Rahia	none	1964
Mere	female	Rewi	Rahia	none	1965
Pita	male	unknown	unknown	Hiniera	1960
Moeraa	female	Pita	Hiniera	none	1986
Huia	female	Pita	Hiniera	none	1987
Irihaapeti	female	Pita	Hiniera	none	1988

The Whanau Program — Database Facts

- We start by encoding the family as facts in the Prolog database.

```
% person(name,  sex,     father,mother,spouse, birth-year)

person(hoone,   male,    unkn1, unkn5, rita,     1910).
person(rita,    female,  unkn2, unkn6, hoone,    1915).
person(ranginui,male,    unkn3, unkn7, reremoana,1915).
person(reremoana, female,unkn4, unkn8, ranginui, 1916).

person(rewi,    male,    hoone, rita, reremoana, 1935).
person(rahia,   female,ranginui,reremoana, rita, 1916).

person(hata,    male,    rewi,  rahia,  none,    1957).
person(kiri,    female,  rewi,  rahia   none,    1959).
```

The Whanau Program — Database Facts. . .

```
% person(name,  sex,     father,mother,spouse,   birth-yea
person(hiniera, female,  rewi,  rahia,  pita,   1960).
person(anaru,   male,    rewi,  rahia,  none,   1962).
person(rapeta,  male,    rewi,  rahia,  none,   1964).
person(mere,    female,  rewi,  rahia,  none,   1965).
person(pita,    male,    unkn9, unkn10, hiniera,1960).

person(moeraa,  female,   hiniera,  pita, none, 1986).
person(huia,    female,   hiniera,  pita, none, 1987).
person(irihaapeti, female,hiniera,  pita, none, 1988).
```

Whanau — Auxiliary predicates

- We introduce some auxiliary predicates to extract information from the database.

```
% Auxiliary predicates
gender(X, G) :- person(X, G, _, _, _, _).
othergender(male, female).
othergender(female, male).
female(X) :- gender(X, female).
male(X)   :- gender(X, male).
```

Whanau — Family Relationships

- We next write some predicates that computes common family relationships.

```
% Is Y the <operator> of X?
wife(X, Y)     :- person(X, male, _, _, Y, _).
husband(X, Y) :- person(X, female, _, _, Y, _).
spouse(X, Y)  :- wife(X, Y).
spouse(X, Y)  :- husband(X, Y).
parent(X, Y)  :- person(X, _,Y, _, _, _).
parent(X, Y)  :- person(X, _,  _, Y, _, _).
son(X, Y)     :- person(Y, male, X, _, _, _).
son(X, Y)     :- person(Y, male, _, X, _, _).
daughter(X, Y):- person(Y, female, X, _, _, _).
daughter(X, Y):- person(Y, female, _, X, _, _).
child(X, Y)   :- son(X, Y).
child(X, Y)   :- daughter(X, Y)
```

Whanau — Family Relationships. . .

- Some of the following are left as an exercise:

```
% Is X older than Y?
older(X,Y) :-
      person(X, _, _, _, _,Xyear),
      person(Y, _, _, _, _,Yyear),
      Yyear > Xyear.

% Is Y a sibling of X of the gender G?
sibling(X, Y, G) :- <left as an exercise>.

% Is Y one of X's older siblings of gender G?
oldersibling(X,Y,G) :- <left as an exercise>.

% Is Y one of X's older/younger siblings of either gender?
oldersibling(X,Y) :- <left as an exercise>.
```

Whanau — Family Relationships. . .

```
youngersibling(X,Y) :- <left as an exercise>.

% Is Y an ancestor of X of gender G?
ancestor(X,Y,G) :- <left as an exercise>.

% Is Y an older relative of X of gender G?
olderrelative(X,Y,G) :-
   ancestor(X, Y, G).
olderrelative(X,Y,G) :-
   ancestor(X, Z, _),
   sibling(Y, Z, G).

% Is Y a sibling of X of his/her opposite gender?
siblingofothersex(X, Y) :- <left as an exercise>.
```

The Whanau Program — Calls

- We can now finally write the predicate `calls(X,Y,T)` which computes all the ways `T` in which `X` can address `Y`.

```
% Me.
calls(X, X, au).

% Parents.
calls(X,Y,paapaa) :- person(X, _,Y, _, _, _).
calls(X,Y,maamaa) :- person(X, _, _,Y, _, _).

% Oldest/youngest sibling of same sex.
calls(X, Y, tuakana) :-
   gender(X, G), eldestsibling(X, Y, G).
calls(X, Y, teina) :-
   gender(X, G), youngestsibling(X, Y, G).
```

The Whanau Program — Calls...

```
% Siblings of other sex.
calls(X, Y, tungaane) :- <left as an exercise>.
calls(X, Y, tuahine) :- <left as an exercise>.
calls(X, Y, tipuna) :-  <left as an exercise>.

% Sons and daughters.
calls(X, Y, tama)   :-  <left as an exercise>.
calls(X, Y, tamahine) :-  <left as an exercise>.

% Oldest/youngest child.
calls(X, Y, maataamua) :- <left as an exercise>.
calls(X, Y, pootiki) :- <left as an exercise>.

% Child-in-law.
calls(X, Y, hunaonga) :-  <left as an exercise>.
```

Readings and References

- Read Clocksin-Mellish, Chapter 2.

Summary

Prolog So Far

- Prolog terms:
  - atoms (`a, 1, 3.14`)
  - structures

    `guitar(ovation, 1111, 1975)`
- Infix expressions are abbreviations of “normal” Prolog terms:

`infix`	`prefix`
`a + b`	`+(a, b)`
`a + b* c`	`+(a, *(b, c))`

CSc 372

Comparative Programming Languages

24 : Prolog — Structures

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

Prolog Structures

- Aka, structured or compound objects
- An object with several components.
- Similar to Pascal's Record-type, C's struct, Haskell's tuples.
- Used to group things together.
- $\overbrace{\texttt{course}}^{\text{functor}}\overbrace{\texttt{(prolog,chris,mon,11)}}^{\text{arguments}}$
- The arity of a functor is the number of arguments.

Example – Course

Structures – Courses

- Below is a database of courses and when they meet. Write the following predicates:
  - `lectures(Lecturer, Day)` succeeds if `Lecturer` has a class on `Day`.
  - `duration(Course, Length)` computes how many hours `Course` meets.
  - `occupied(Room, Day, Time)` succeeds if `Room` is being used on `Day` at `Time`.

```
% course(class, meetingtime, prof, hall).
course(c231, time(mon,4,5), cc, plt1).
course(c231, time(wed,10,11), cc, plt1).
course(c231, time(thu,4,5), cc, plt1).
course(c363, time(mon,11,12), cc, slt1).
course(c363, time(thu,11,12), cc, slt1).
```

Structures – Courses…

```
lectures(Lecturer, Day) :-
   course(Course, time(Day,_,_), Lecturer, _).

duration(Course, Length) :-
   course(Course,
         time(Day,Start,Finish), Lec, Loc),
   Length is Finish - Start.

occupied(Room, Day, Time) :-
   course(Course,
         time(Day,Start,Finish), Lec, Room),
   Start =< Time,
   Time =< Finish.
```

Structures – Courses. . .

```
course(c231, time(mon,4,5), cc, plt1).
course(c231, time(wed,10,11), cc, plt1).
course(c231, time(thu,4,5), cc, plt1).
course(c363, time(mon,11,12), cc, slt1).
course(c363, time(thu,11,12), cc, slt1).
```

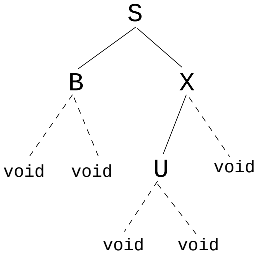
```
?- occupied(slt1, mon, 11).
yes
?- lectures(cc, mon).
yes
```

Example – Binary Trees

Binary Trees

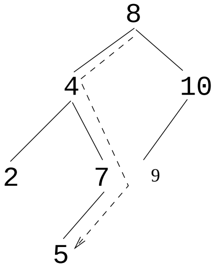
- We can represent trees as nested structures:

```
tree(Element, Left, Right)

tree(s,
  tree(b, void, void),
  tree(x,
    tree(u, void, void),
    void).
```



Binary Search Trees

- Write a predicate `member(T,x)` that succeeds if `x` is a member of the binary search tree `T`:



```
atree(
  tree(8,
    tree(4,
      tree(2,void,void),
      tree(7,
        tree(5,void,void),
        void)),
    tree(10,
      tree(9,void,void),
      void))).
```

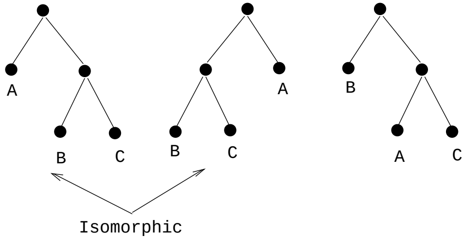
```
?- atree(T),tree_member(5,T).
```

Binary Search Trees. . .

```
tree_member(X, tree(X,_, _)).
tree_member(X, tree(Y,Left,_)) :-
   X < Y,
   tree_member(Y, Left).
tree_member(X, tree(Y,_,Right)) :-
   X > Y,
   tree_member(Y, Right).
```

Binary Trees – Isomorphism

Tree isomorphism:



Two binary trees T_1 and T_2 are isomorphic if T_2 can be obtained by reordering the branches of the subtrees of T_1.

- Write a predicate `tree_iso(T1, T2)` that succeeds if the two trees are isomorphic.

Binary Trees – Isomorphism. . .

```
tree_iso(void, void).

tree_iso(tree(X, L1, R1), tree(X, L2, R2)) :-
   tree_iso(L1, L2), tree_iso(R1, R2).

tree_iso(tree(X, L1, R1), tree(X, L2, R2)) :-
   tree_iso(L1, R2), tree_iso(R1, L2).
```

1. Check if the roots of the current subtrees are identical;
2. Check if the subtrees are isomorphic;
3. If they are not, backtrack, swap the subtrees, and again check if they are isomorphic.

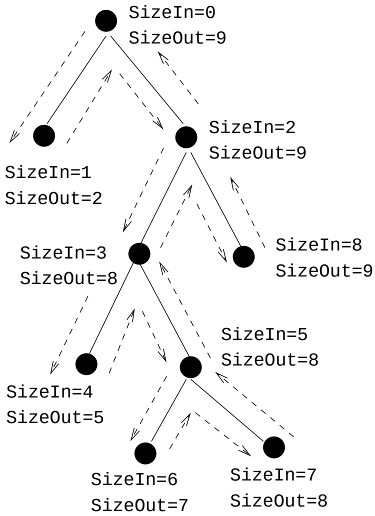
Binary Trees – Counting Nodes

- Write a predicate `size_of_tree(Tree,Size)` which computes the number of nodes in a tree.

```
size_of_tree(Tree, Size) :-
   size_of_tree(Tree, 0, Size).

size_of_tree(void, Size, Size).
size_of_tree(tree(_, L, R), SizeIn, SizeOut) :-
   Size1 is SizeIn + 1,
   size_of_tree(L, Size1, Size2),
   size_of_tree(R, Size2, SizeOut).
```

- We use a so-called accumulator pair to pass around the current size of the tree.

Binary Trees – Counting Nodes. . .

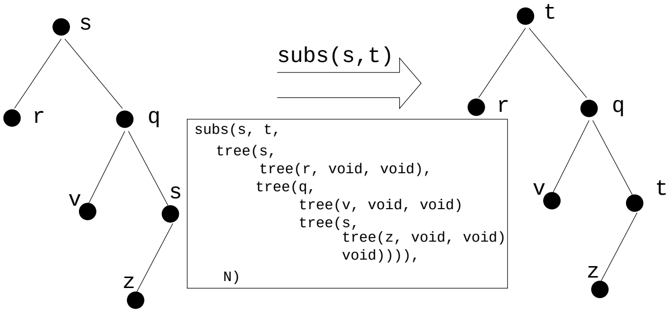


Binary Trees – Tree Substitution

- Write a predicate `subs(T1,T2,Old,New)` which replaces all occurences of `Old` with `New` in tree `T1`:

```
subs(X, Y, void, void).
subs(X, Y, tree(X, L1, R1), tree(Y, L2, R2)) :-
   subs(X, Y, L1, L2),
   subs(X, Y, R1, R2).
subs(X, Y, tree(Z, L1, R1), tree(Z, L2, R2)) :-
   X =\= Y, subs(X, Y, L1, L2),
   subs(X, Y, R1, R2).
```

Binary Trees – Tree Substitution. . .



```
subs(s, t,
  tree(s,
        tree(r, void, void),
       tree(q,
            tree(v, void, void)
            tree(s,
                 tree(z, void, void)
                 void)))),
   N)
```

Symbolic Differentiation

Symbolic Differentiation

$$\frac{\mathrm{d}c}{\mathrm{d}x} = 0 \tag{1}$$

$$\frac{\mathrm{d}x}{\mathrm{d}x} = 1 \tag{2}$$

$$\frac{\mathrm{d}(U^c)}{\mathrm{d}x} = cU^{c-1}\frac{\mathrm{d}U}{\mathrm{d}x} \tag{3}$$

$$\frac{\mathrm{d}(-U)}{\mathrm{d}x} = -\frac{\mathrm{d}U}{\mathrm{d}x} \tag{4}$$

$$\frac{\mathrm{d}(U+V)}{\mathrm{d}x} = \frac{\mathrm{d}U}{\mathrm{d}x} + \frac{\mathrm{d}V}{\mathrm{d}x} \tag{5}$$

$$\frac{\mathrm{d}(U-V)}{\mathrm{d}x} = \frac{\mathrm{d}U}{\mathrm{d}x} - \frac{\mathrm{d}U}{\mathrm{d}x} \tag{6}$$

Symbolic Differentiation...

$$\frac{\mathrm{d}(cU)}{\mathrm{d}x} = c\frac{\mathrm{d}U}{\mathrm{d}x} \tag{7}$$

$$\frac{\mathrm{d}(UV)}{\mathrm{d}x} = U\frac{\mathrm{d}V}{\mathrm{d}x} + V\frac{\mathrm{d}U}{\mathrm{d}x} \tag{8}$$

$$\frac{\mathrm{d}(\frac{U}{V})}{\mathrm{d}x} = \frac{V\frac{\mathrm{d}U}{\mathrm{d}x} - U\frac{\mathrm{d}V}{\mathrm{d}x}}{V^2} \tag{9}$$

$$\frac{\mathrm{d}(\ln U)}{\mathrm{d}x} = U^{-1}\frac{\mathrm{d}U}{\mathrm{d}x} \tag{10}$$

$$\frac{\mathrm{d}(\sin(U))}{\mathrm{d}x} = \frac{\mathrm{d}U}{\mathrm{d}x}\cos(U) \tag{11}$$

$$\frac{\mathrm{d}(\cos(U))}{\mathrm{d}x} = -\frac{\mathrm{d}U}{\mathrm{d}x}\sin(U) \tag{12}$$

Symbolic Differentiation...

$$\frac{\mathrm{d}c}{\mathrm{d}x} = 0 \tag{1}$$

$$\frac{\mathrm{d}x}{\mathrm{d}x} = 1 \tag{2}$$

$$\frac{\mathrm{d}(U^c)}{\mathrm{d}x} = cU^{c-1}\frac{\mathrm{d}U}{\mathrm{d}x} \tag{3}$$

```
deriv(C, X, 0) :- number(C).

deriv(X, X, 1).

deriv(U ^C, X, C * U ^L * DU) :-
        number(C), L is C - 1, deriv(U, X, DU).
```

Symbolic Differentiation…

$$\frac{\mathrm{d}(-U)}{\mathrm{d}x} = -\frac{\mathrm{d}U}{\mathrm{d}x} \tag{4}$$

$$\frac{\mathrm{d}(U+V)}{\mathrm{d}x} = \frac{\mathrm{d}U}{\mathrm{d}x} + \frac{\mathrm{d}V}{\mathrm{d}x} \tag{5}$$

```
deriv(-U, X, -DU) :-
   deriv(U, X, DU).

deriv(U+V, X, DU + DV) :-
   deriv(U, X, DU),
   deriv(V, X, DV).
```

Symbolic Differentiation…

$$\frac{\mathrm{d}(U - V)}{\mathrm{d}x} = \frac{\mathrm{d}U}{\mathrm{d}x} - \frac{\mathrm{d}V}{\mathrm{d}x} \tag{6}$$

$$\frac{\mathrm{d}(cU)}{\mathrm{d}x} = c\frac{\mathrm{d}U}{\mathrm{d}x} \tag{7}$$

```
deriv(U-V, X, __________) :-
   <left as an exercise>

deriv(C*U, X, __________) :-
   <left as an exercise>
```

Symbolic Differentiation...

$$\frac{\mathrm{d}(UV)}{\mathrm{d}x} = U\frac{\mathrm{d}V}{\mathrm{d}x} + V\frac{\mathrm{d}U}{\mathrm{d}x} \tag{8}$$

$$\frac{\mathrm{d}(\frac{U}{V})}{\mathrm{d}x} = \frac{V\frac{\mathrm{d}U}{\mathrm{d}x} - U\frac{\mathrm{d}V}{\mathrm{d}x}}{V^2} \tag{9}$$

```
deriv(U*V, X, __________) :-
   <left as an exercise>

deriv(U/V, X, __________) :-
   <left as an exercise>
```

Symbolic Differentiation...

$$\frac{\mathrm{d}(\ln U)}{\mathrm{d}x} = U^{-1}\frac{\mathrm{d}U}{\mathrm{d}x} \tag{10}$$

$$\frac{\mathrm{d}(\sin(U))}{\mathrm{d}x} = \frac{\mathrm{d}U}{\mathrm{d}x}\cos(U) \tag{11}$$

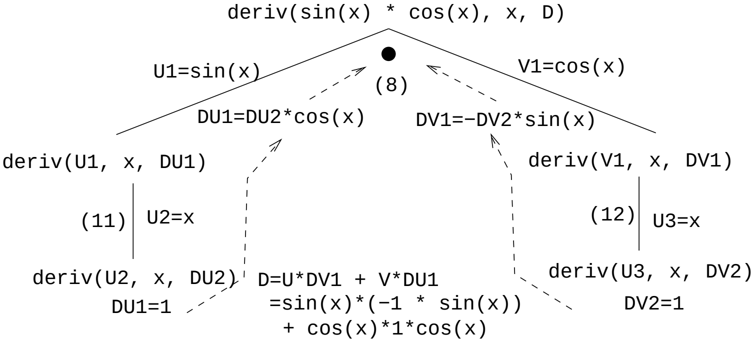
$$\frac{\mathrm{d}(\cos(U))}{\mathrm{d}x} = -\frac{\mathrm{d}U}{\mathrm{d}x}\sin(U) \tag{12}$$

```
deriv(log(U), X, _______) :- <left as an exercise>

deriv(sin(U), X, _______) :- <left as an exercise>

deriv(cos(U), X, _______) :- <left as an exercise>
```

Symbolic Differentiation...

```
?- deriv(x, x, D).
   D = 1

?- deriv(sin(x), x, D).
   D = 1*cos(x)

?- deriv(sin(x) + cos(x), x, D).
   D = 1*cos(x)+ (-1*sin(x))

?- deriv(sin(x) * cos(x), x, D).
   D = sin(x)* (-1*sin(x)) +cos(x)* (1*cos(x))

?- deriv(1 / x, x, D).
   D = (x*0-1*1)/ (x*x)
```

Symbolic Differentiation. . .



deriv(sin(x) * cos(x), x, D)

U1=sin(x)

(8)

V1=cos(x)

DU1=DU2*cos(x)

DV1=-DV2*sin(x)

deriv(U1, x, DU1)

deriv(V1, x, DV1)

(11) U2=x

(12) U3=x

deriv(U2, x, DU2)

DU1=1

D=U*DV1 + V*DU1
=sin(x)*(-1 * sin(x))
+ cos(x)*1*cos(x)

deriv(U3, x, DV2)

DV2=1

Symbolic Differentiation...

```
?- deriv(1/sin(x), x, D).
   D = (sin(x)*0-1* (1*cos(x)))+(sin(x)*sin(x))

?- deriv(x ^3, x, D).
   D = 1*3*x^2

?- deriv(x^3 + x^2 + 1, x, D).
   D = 1*3*x^2+1*2*x^1+0

?- deriv(3 * x ^3, x, D).
   D = 3* (1*3*x^2)+x^3*0

?- deriv(4* x ^3 + 4 * x^2 + x - 1, x, D).
   D = 4* (1*3*x^2)+x^3*0+(4* (1*2*x^1)+x^2*0)+1-0
```

Readings and References

- Read Clocksin-Mellish, Sections 2.1.3, 3.1.

Summary

Prolog So Far…

- Prolog terms:
    - atoms (`a, 1, 3.14`)
    - structures

      `guitar(ovation, 1111, 1975)`
- Infix expressions are abbreviations of "normal" Prolog terms:

infix	prefix
`a + b`	`+(a, b)`
`a + b∗ c`	`+(a, ∗(b, c))`

CSc 372

Comparative Programming Languages

25 : Prolog — Matching

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

Unification & Matching

- So far, when we've gone through examples, I have said simply that when trying to satisfy a goal, Prolog searches for a matching rule or fact.
- What does this mean, to match?
- Prolog's matching operator or `=`. It tries to make its left and right hand sides the same, by assigning values to variables.
- Also, there's an implicit `=` between arguments when we try to match a query

  ```
  ?- f(x,y)
  ```

  to a rule

  ```
  f(A,B) :- ....
  ```

Matching Examples

The rule:

```
deriv(U ^C, X, C * U ^L * DU) :-
        number(C), L is C - 1,
        deriv(U, X, DU).
```

```
?- deriv(x ^3, x, D).
   D = 1*3*x^2
```

The goal:

- `x ^3` matches `U ^C`
    - $x = U, C = 3$
- `x` matches `X`
- `D` matches `C * U ^L * DU`

Matching Examples. . .

```
deriv(U+V, X, DU + DV) :-
   deriv(U, X, DU),
   deriv(V, X, DV).

?- deriv(x^3 + x^2 + 1, x, D).
   D = 1*3*x^2+1*2*x^1+0
```

- `x ^3 + x^2 + 1` matches `U + V`
    - `x ^3 + x^2` is bound to `U`
    - `1` is bound to `V`

Matching Algorithm

> *Can two terms A and F be "made identical," by assigning values to their variables?*

Two terms A and F match if

1. they are identical atoms
2. one or both are uninstantiated variables
3. they are terms $A = f_A(a_1, \cdots, a_n)$ and $F = f_F(f_1, \cdots, f_m)$, and
    1. the arities are the same ($n = m$)
    2. the functors are the same ($f_A = f_F$)
    3. the arguments match ($a_i \equiv f_i$)

Matching – Examples

A	F	$A \equiv F$	variable subst.
a	a	yes	
a	b	no	
sin(X)	sin(a)	yes	$\theta = \{\mathrm{X}{=}\mathrm{a}\}$
sin(a)	sin(X)	yes	$\theta = \{\mathrm{X}{=}\mathrm{a}\}$
cos(X)	sin(a)	no	
sin(X)	sin(cos(a))	yes	$\theta = \{\mathrm{X}{=}\mathrm{cos(a)}\}$

Matching – Examples. . .

A	F	$A \equiv F$	variable subst.
likes(c, X)	likes(a, X)	no	
likes(c, X)	likes(c, Y)	yes	$\theta = \{\mathrm{X{=}Y}\}$
likes(X, X)	likes(c, Y)	yes	$\theta = \{\mathrm{X{=}c,\ X{=}Y}\}$
likes(X, X)	likes(c, _)	yes	$\theta = \{\mathrm{X{=}c,\ X{=}\_47}\}$
likes(c, a(X))	likes(V, Z)	yes	$\theta = \{\mathrm{V{=}c,Z{=}a(X)}\}$
likes(X, a(X))	likes(c, Z)	yes	$\theta = \{\mathrm{X{=}c,Z{=}a(X)}\}$

Matching Consequences

Consequences of Prolog Matching:

- An uninstantiated variable will match any object.
- An integer or atom will match only itself.
- When two uninstantiated variables match, they *share*:
  - When one is instantiated, so is the other (with the same value).
- Backtracking undoes all variable bindings.

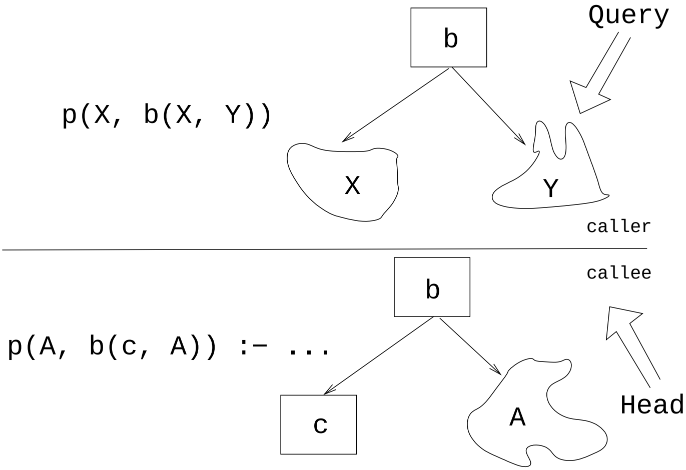
Matching Algorithm

```
FUNC Unify (A, F: term) :  BOOL;
   IF Is_Var(F) THEN Instantiate F to A
   ELSIF Is_Var(A) THEN Instantiate A to F
   ELSIF Arity(F)$\neq$Arity(A) THEN RETURN FALSE
   ELSIF Functor(F)$\neq$Functor(A) THEN RETURN FALSE
   ELSE
      FOR each argument $i$ DO
         IF NOT Unify(A($i$), F($i$)) THEN
            RETURN FALSE
   RETURN TRUE;
```

Visualizing Matching

- From *Prolog for Programmers*, Kluzniak & Szpakowicz, page 18.
- Assume that during the course of a program we attempt to match the goal `p(X, b(X, Y))` with a clause *C*, whose head is `p(X, b(X, y))`.
- First we'll compare the arity and name of the functors. For both the goal and the clause they are 2 and `p`, respectively.

Visualizing Matching. . .



`p(X, b(X, Y))`

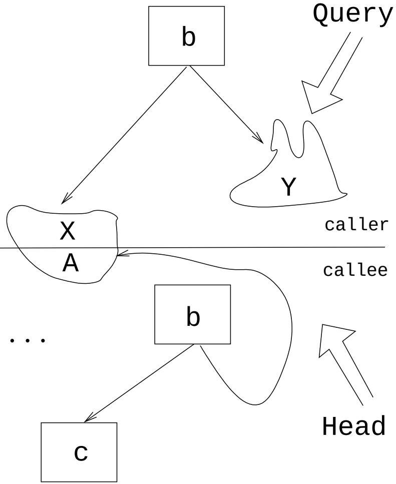
`p(A, b(c, A)) :- ...`

Visualizing Matching. . .

- The second step is to try to unify the first argument of the goal (`X`) with the first argument of the clause head (`A`).
- They are both variables, so that works OK.
- From now on `A` and `X` will be treated as identical (they are in the list of variable substitutions θ).

Visualizing Matching. . .

$\texttt{p(}\boxed{\texttt{X}}\texttt{, b(X, Y))}$



$\texttt{p(}\boxed{\texttt{A}}\texttt{, b(c, A)) :- ...}$

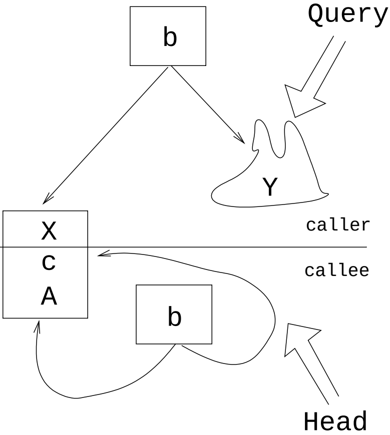
$\theta = \{A = X\}$

Visualizing Matching. . .

- Next we try to match the second argument of the goal (`b(X, Y)`) with the second argument of the clause head (`b(c, A)`).
- The arities and the functors are the same, so we go on to to try to match the arguments.
- The first argument in the goal is `X`, which is matched by the first argument in the clause head (`c`). I.e., `X` and `c` are now treated as identical.

Visualizing Matching. . .

$\texttt{p(X, b(}\boxed{\texttt{X}}\texttt{, Y))}$



$\texttt{p(A, b(}\boxed{\texttt{c}}\texttt{, A)) :- ...}$

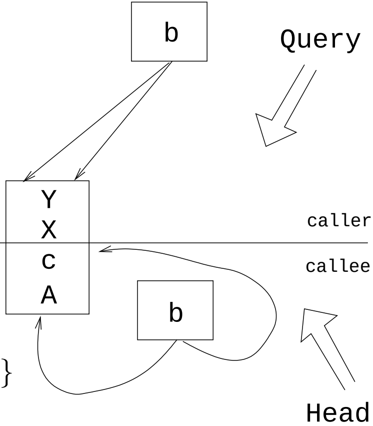
$$\theta = \{A = X, X = c\}$$

Visualizing Matching. . .

- Finally, we match `A` and `Y`. Since `A=X` and `X=c`, this means that `Y=c` as well.

Visualizing Matching. . .

$\texttt{p(X, b(X, }\boxed{\texttt{Y}}\texttt{))}$



$\texttt{p(A, b(c, }\boxed{\texttt{A}}\texttt{)) :- ...}$

$\theta = \{A = X, X = c, A = Y\}$

Summary

Readings and References

- Read Clocksin-Mellish, Sections 2.4, 2.6.3.

Prolog So Far…

- A term is either a
  - a constant (an atom or integer)
  - a variable
  - a structure
- Two terms *match* if
  - there exists a variable substitution θ which makes the terms identical.
- Once a variable becomes instantiated, it stays instantiated.
- Backtracking *undoes* variable instantiations.
- Prolog searches the database sequentially (from top to bottom) until a matching clause is found.

CSc 372

Comparative Programming Languages

26 : Prolog — Execution

Department of Computer Science
University of Arizona

collberg@gmail.com

Execution

Executing Prolog

- Now that we know about matching, we can take a closer look at how Prolog tries to satisfy goals.
- In general, to solve a goal

  $$G = G_1, G_2, \cdots , G_m,$$

  Prolog will first try to solve the sub-goal G_1.
- It solves a sub-goal G_1 it will look for a rule

  $$H_i \text{ :- } B_1, \cdots , B_n$$

  in the database, such that G_1 and H_i will match.
- Any variable substitutions resulting from the match will be stored in a variable θ.

Executing Prolog. . .

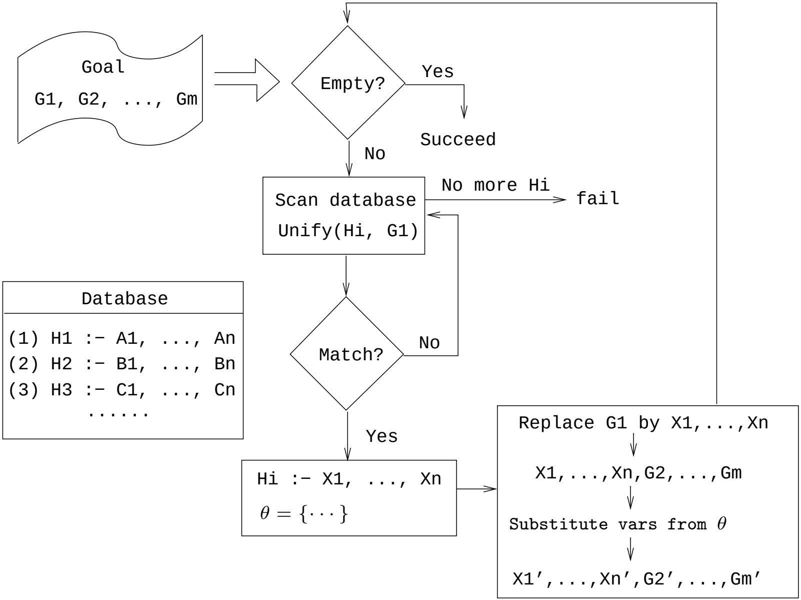
- A new goal will be constructed by replacing G_1 with $B_1, \cdots, B_n$, yielding

  $$G' = B_1, \cdots, B_n, G_2, \cdots, G_m.$$

  If $n = 0$ the new goal will be shorter and we'll be one step closer to a solution to G!
- Any new variable bindings from θ are applied to the new goal, yielding G''.
- We recursively try to find a solution to G''.

Executing Prolog...

```
FUNC Execute ($G = G_1, G_2, \cdots, G_m$; Result);
    IF Is_Empty($G$) THEN Result := Yes
    ELSE
        Result := No;
        $i$ := 1;
        WHILE Result=No & $i \leq$ NoOfClauses DO
            Clause := $H_i$ :- $B_1, \cdots, B_n$;
            IF Unify($G_1$, Clause, $\theta$) THEN
                $G'$ := $B_1, \cdots, B_n, G_2, \cdots, G_m$;
                $G''$ := substitute($G'$, $\theta$);
                Execute($G''$, Result);
            ENDIF;
            $i$ := $i + 1$;
        ENDDO
    ENDIF
```


Goal
G1, G2, ..., Gm
Empty?
Yes
Succeed
No
Scan database
Unify(Hi, G1)
No more Hi
fail
Database
(1) H1 :- A1, ..., An
(2) H2 :- B1, ..., Bn
(3) H3 :- C1, ..., Cn
......
Match?
No
Yes
Hi :- X1, ..., Xn
$\theta = \{\cdots\}$
Replace G1 by X1,...,Xn
X1,...,Xn,G2,...,Gm
Substitute vars from θ
X1′,...,Xn′,G2′,...,Gm′

Example

Northern Exposure Example

```
% From the Northern Exposure FAQ
% friend(of, kind(name, regular)).
friend(maggie, person(eve, yes)).
friend(maggie, moose(morty, yes)).
friend(maggie, person(harry, no)).
friend(maggie, person(bruce, no)).
friend(maggie, person(glenn, no)).
friend(maggie, person(dave, no)).
friend(maggie, person(rick, no)).
friend(maggie, person(mike, yes)).
friend(maggie, person(joel, yes)).
```

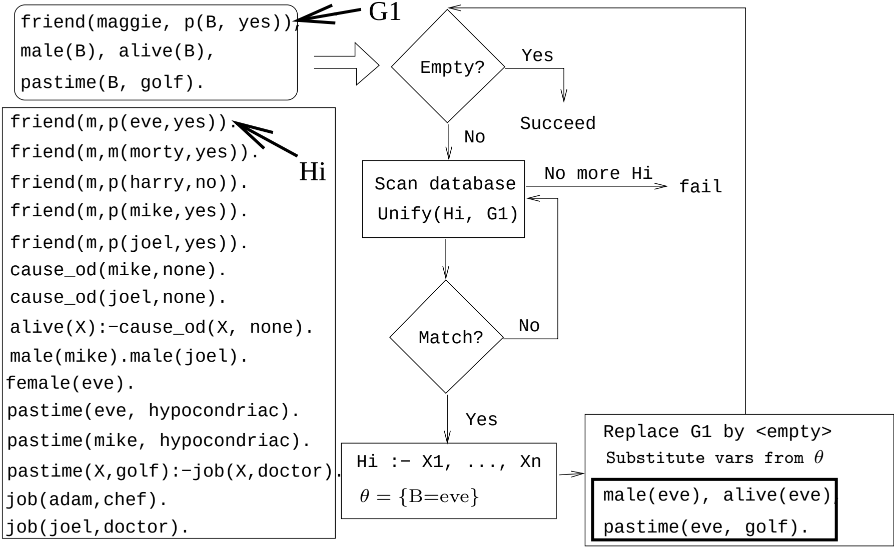
Maggie (Janine Turner)

Northern Exposure Example...

```
cause_of_death(morty, copper_deficiency).
cause_of_death(harry, potato_salad).
cause_of_death(bruce, fishing_accident).
cause_of_death(glenn, missile).
cause_of_death(dave, hypothermia).
cause_of_death(rick, hit_by_satellite).
cause_of_death(mike, none_yet).
cause_of_death(joel, none_yet).

male(morty).  male(harry).  male(bruce).
male(glenn).  male(dave).  male(rick).
male(mike).  male(joel).  female(eve).
```

Northern Exposure Example...

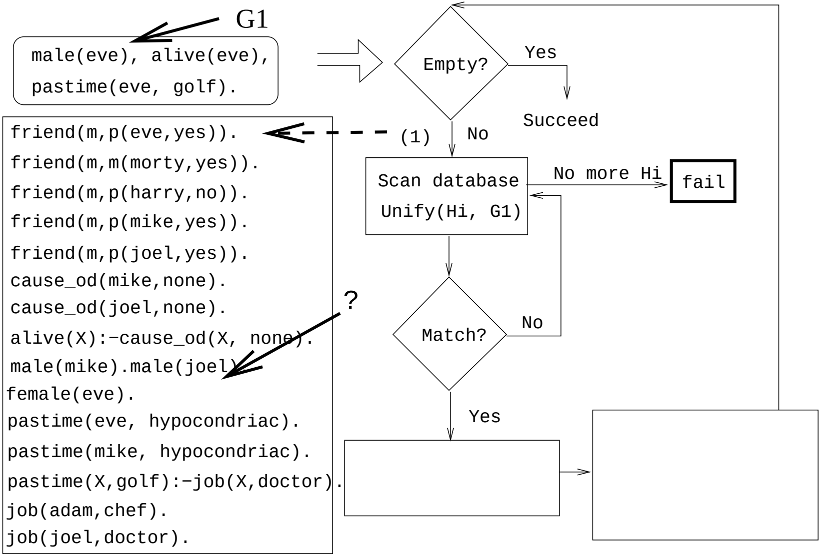
```
alive(X) :- cause_of_death(X, none_yet).

pastime(eve, hypochondria).
pastime(mike, hypochondria).
pastime(X, golf) :- job(X,doctor).

job(mike, lawyer).  job(adam, chef).
job(maggie, pilot).  job(joel, doctor).

?- friend(maggie, person(B, yes)),
      male(B),
      alive(B),
      pastime(B, golf).
```



```
friend(maggie, p(B, yes)),
male(B), alive(B),
pastime(B, golf).
```

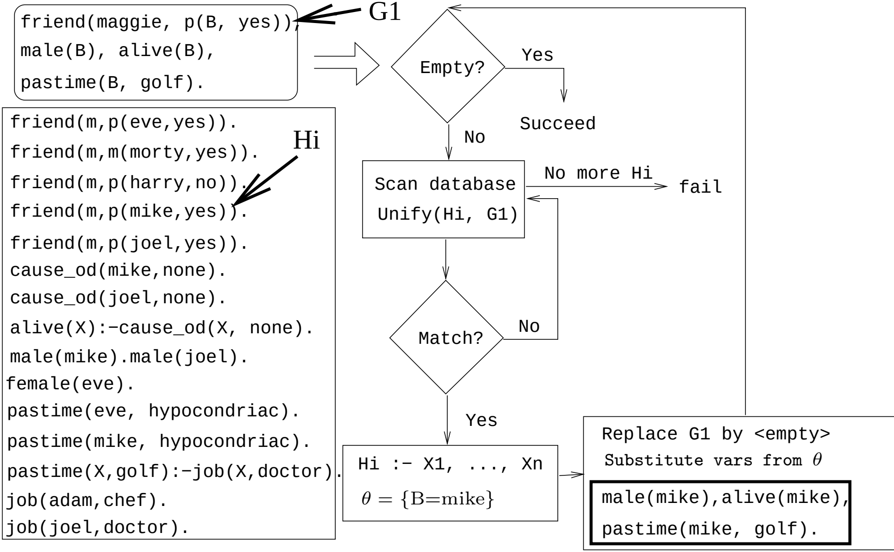
```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypocondriac).
pastime(mike, hypocondriac).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```

```
male(eve), alive(eve),
pastime(eve, golf).
```



```
male(eve), alive(eve),
pastime(eve, golf).
```

```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypocondriac).
pastime(mike, hypocondriac).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```



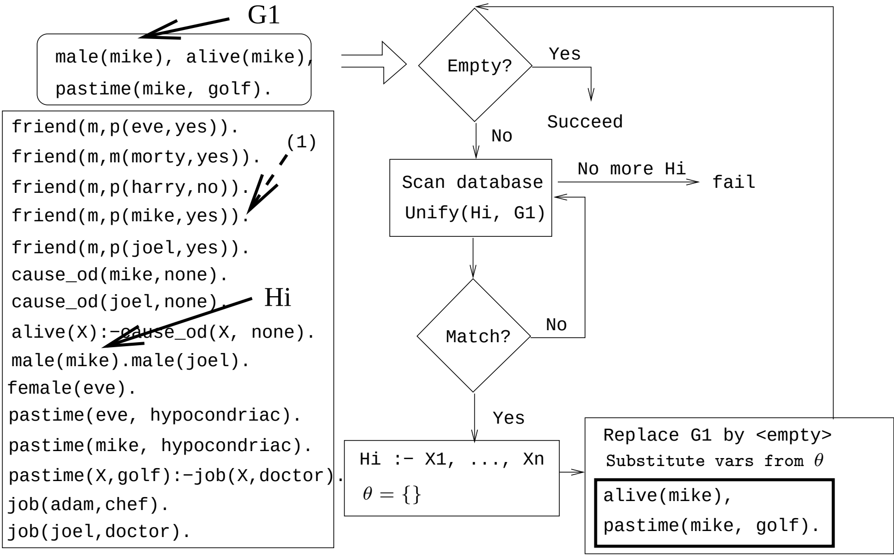
```
friend(maggie, p(B, yes)),
male(B), alive(B),
pastime(B, golf).
```

```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypocondriac).
pastime(mike, hypocondriac).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```

```
Hi :- X1, ..., Xn
```

$\theta = \{\mathrm{B{=}mike}\}$

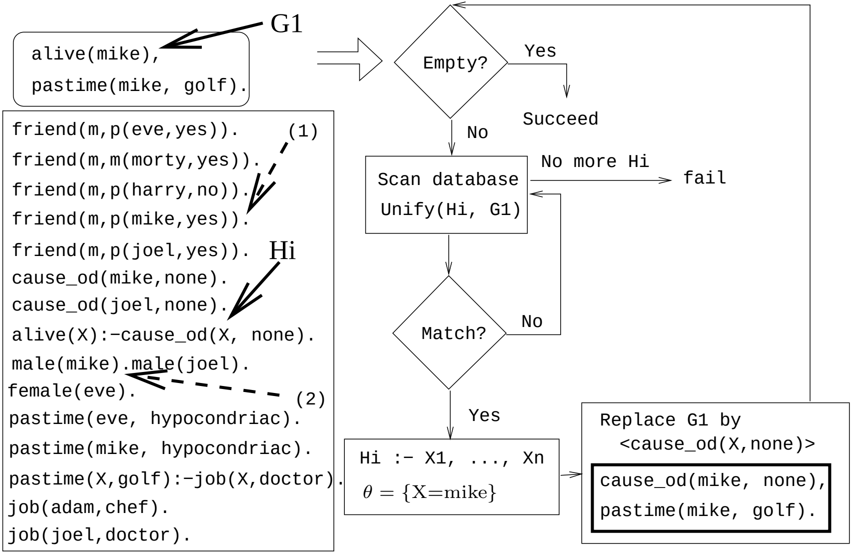
```
male(mike),alive(mike),
pastime(mike, golf).
```



```
male(mike), alive(mike),
pastime(mike, golf).
```

```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypocondriac).
pastime(mike, hypocondriac).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```

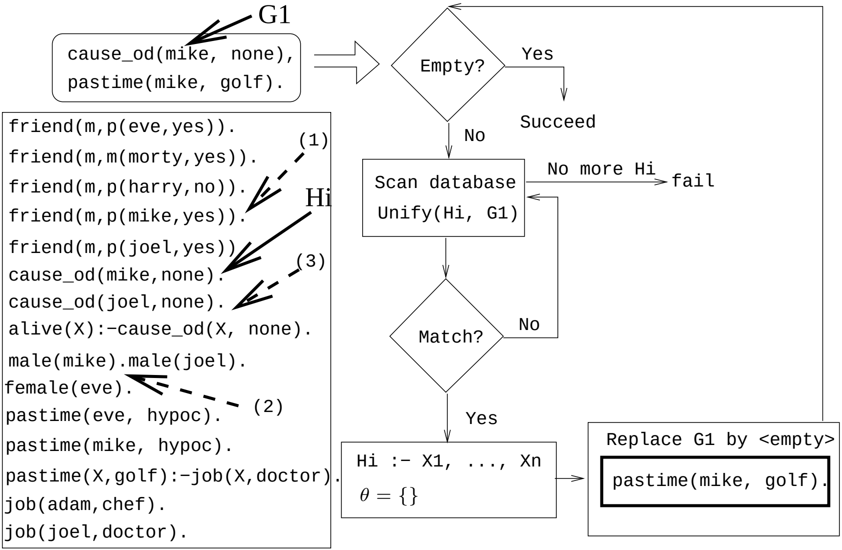
```
alive(mike),
pastime(mike, golf).
```



```
alive(mike),
pastime(mike, golf).
```

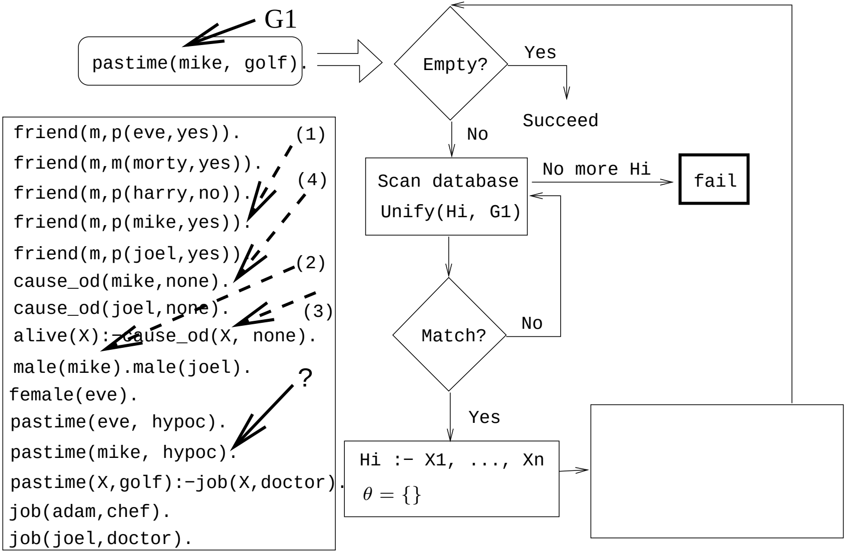
```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):−cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypocondriac).
pastime(mike, hypocondriac).
pastime(X,golf):−job(X,doctor).
job(adam,chef).
job(joel,doctor).
```

```
Replace G1 by
  <cause_od(X,none)>
cause_od(mike, none),
pastime(mike, golf).
```



```
cause_od(mike, none),
pastime(mike, golf).
```

```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes))
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypoc).
pastime(mike, hypoc).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```



```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypoc).
pastime(mike, hypoc).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```

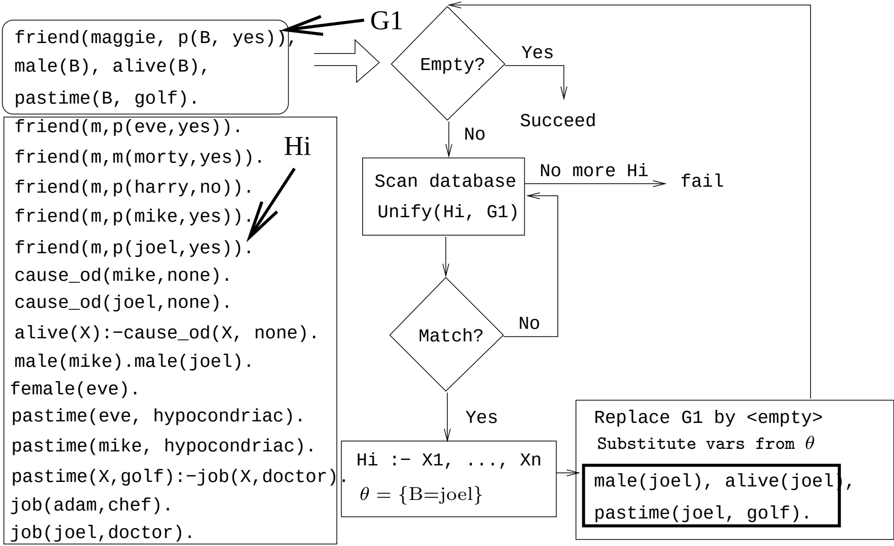
Northern Exposure Example. . .

- We skip a step here.
- `pastime(mike, golf)` unifies with

```
pastime(X, golf) :- job(X, doctor).
```

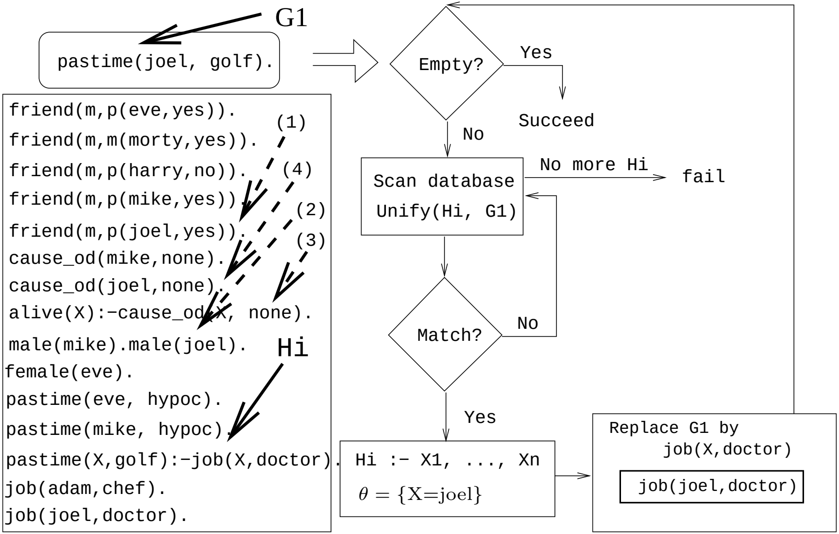
.

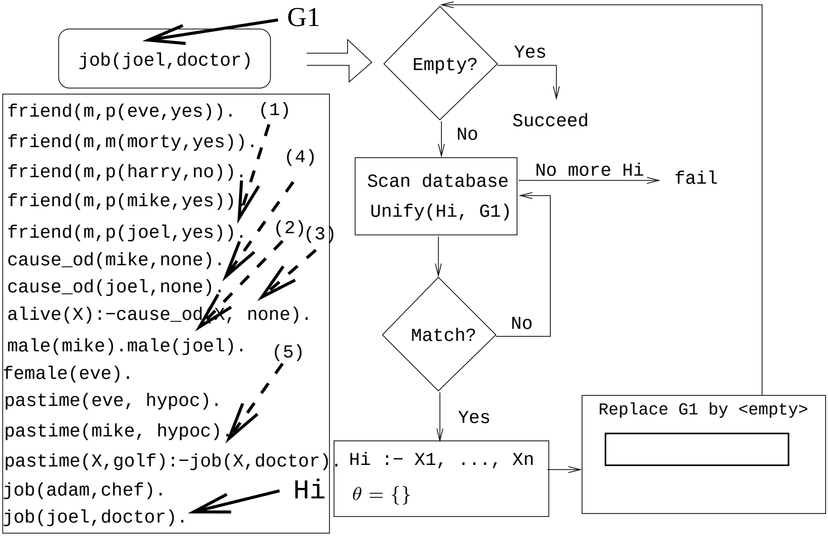
- However, `job(mike, doctor)` fails, and we backtrack all the way up to the original query.



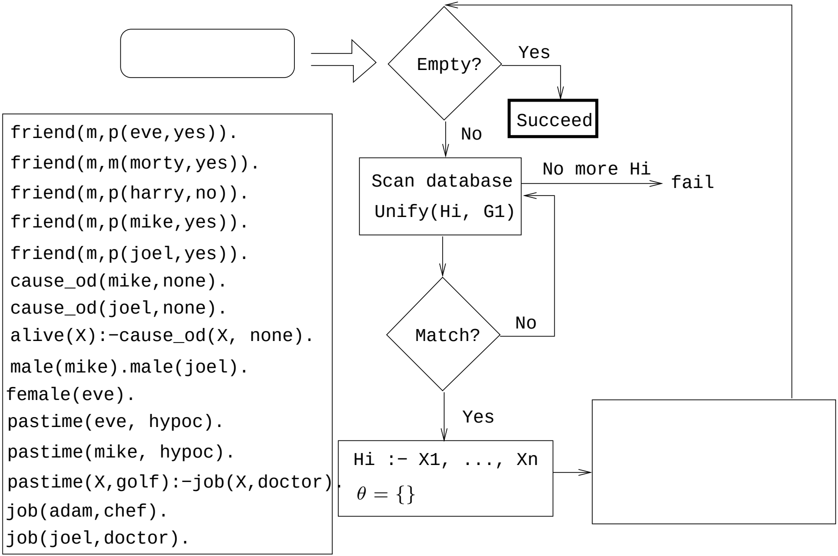
```
friend(maggie, p(B, yes)),
male(B), alive(B),
pastime(B, golf).
```

```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypocondriac).
pastime(mike, hypocondriac).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```


G1
pastime(joel, golf).
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):−cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypoc).
pastime(mike, hypoc).
pastime(X,golf):−job(X,doctor).
job(adam,chef).
job(joel,doctor).
(1)
(4)
(2)
(3)
Hi
Empty?
Yes
Succeed
No
Scan database
Unify(Hi, G1)
No more Hi
fail
Match?
No
Yes
Hi :− X1, ..., Xn
$\theta = \{X{=}\text{joel}\}$
Replace G1 by
job(X,doctor)
job(joel,doctor)



```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes))
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypoc).
pastime(mike, hypoc).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```



```
friend(m,p(eve,yes)).
friend(m,m(morty,yes)).
friend(m,p(harry,no)).
friend(m,p(mike,yes)).
friend(m,p(joel,yes)).
cause_od(mike,none).
cause_od(joel,none).
alive(X):-cause_od(X, none).
male(mike).male(joel).
female(eve).
pastime(eve, hypoc).
pastime(mike, hypoc).
pastime(X,golf):-job(X,doctor).
job(adam,chef).
job(joel,doctor).
```

Readings and References

- Read Clocksin-Mellish, Section 4.1.
- See `http://www.moosefest.org` for information about the annual Moosefest.
- See `http://members.lycos.co.uk/janineturner/engl/index.html` for pictures of Janine Turner, who plays Maggie.
- See `http://home.comcast.net/~mcnotes/mcnotes.html` for show transcripts.

Summary

Prolog So Far…

- A term is either a
    - a constant (an atom or integer)
    - a variable
    - a structure
- Two terms *match* if
    - there exists a variable substitution θ which makes the terms identical.
- Once a variable becomes instantiated, it stays instantiated.
- Backtracking *undoes* variable instantiations.
- Prolog searches the database sequentially (from top to bottom) until a matching clause is found.

CSc 372

Comparative Programming Languages

27 : Prolog — Lists

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

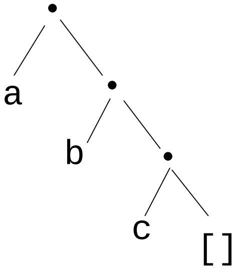
Prolog Lists

Haskell:

```
> 1 :  2 :  3 :  []
[1,2,3]
```

Prolog:

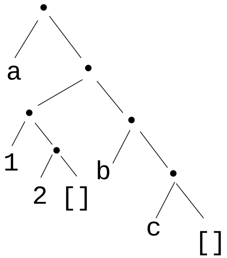
```
?- L = .(a, .(b, .(c, []))) 
     L = [a, b, c]
```



- Both Haskell and Prolog build up lists using cons-cells.
- In Haskell the cons-operator is `:`, in Prolog `.`.

Prolog Lists. . .

```
?- L = .(a, .(.(1, .(2, [])), .(b, .(c, []))))
      L = [a, [1, 2], b, c]
```



- Unlike Haskell, Prolog lists can contain elements of arbitrary type.

Matching Lists – `[Head | Tail]`

A	F	$A \equiv F$	variable subst.
`[]`	`[]`	yes	
`[]`	`a`	no	
`[a]`	`[]`	no	
`[[]]`	`[]`	no	
`[a | [b, c]]`	`L`	yes	`L=[a,b,c]`
`[a]`	`[H | T]`	yes	`H=a, T=[]`

Matching Lists – `[Head | Tail]`...

A	F	$A \equiv F$	variable subst.
`[a, b, c]`	`[H	T]`	yes	`H=a,T=[b,c]`
`[a, [1, 2]]`	`[H	T]`	yes	`H=a, T=[[1, 2]]`
`[[1, 2], a]`	`[H	T]`	yes	`H=[1,2], T=[a]`
`[a, b, c]`	`[X, Y, c]`	yes	`X=a, Y=c`
`[a, Y, c]`	`[X, b, Z]`	yes	`X=a, Y=b, Z=c`
`[a, b]`	`[X, c]`	no	

Member

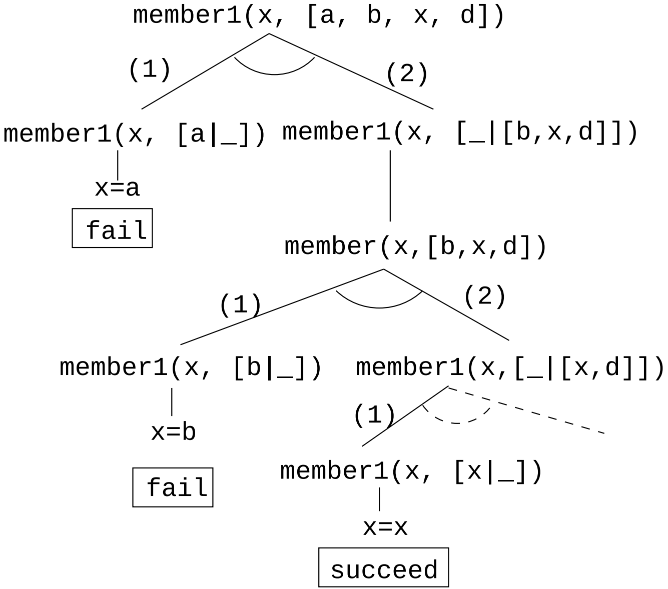
Prolog Lists — Member

```
(1)   member1(X, [Y|_]) :- X = Y.
(2)   member1(X, [_|Y]) :- member1(X, Y).

(1)   member2(X, [X|_]).
(2)   member2(X, [_|Y]) :- member2(X, Y).

(1)   member3(X,[Y|Z]) :- X = Y; member3(X,Z).
```

Prolog Lists — Member...

```
?- member(x, [a, b, c, x, f]).
       yes

?- member(x, [a, b, c, f]).
       no

?- member(x, [a, [x, y], f]).
       no

?- member(Z, [a, [x, y], f]).
       Z = a
       Z = [x, y]
       Z = f
```

Prolog Lists — Member. . .



member1(x, [a, b, x, d])

(1)

(2)

member1(x, [a|_])

x=a

fail

member1(x, [_|[b,x,d]])

member(x,[b,x,d])

(1)

(2)

member1(x, [b|_])

x=b

fail

member1(x,[_|[x,d]])

(1)

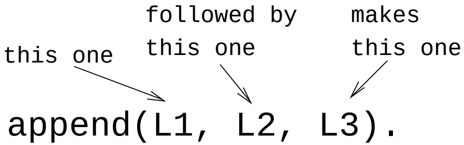
member1(x, [x|_])

x=x

succeed

Append

Prolog Lists — Append



this one

followed by this one

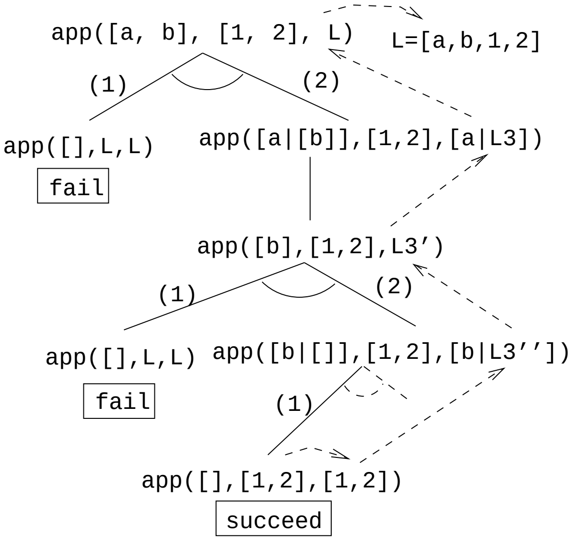
makes this one

```
append(L1, L2, L3).
```

```
(1) append([], L, L)
(2) append([X|L1], L2, [X|L3]) :-
      append(L1, L2, L3).
```

1. Appending L onto an empty list, makes L.
2. To append L_2 onto L_1 to make L_3
    1. Let the first element of L_1 be the first element of L_3.
    2. Append L_2 onto the rest of L_1 to make the rest of L_3.

Prolog Lists — Append. . .



app([a, b], [1, 2], L)

L=[a,b,1,2]

(1)

(2)

app([],L,L)

fail

app([a|[b]],[1,2],[a|L3])

app([b],[1,2],L3′)

(1)

(2)

app([],L,L)

fail

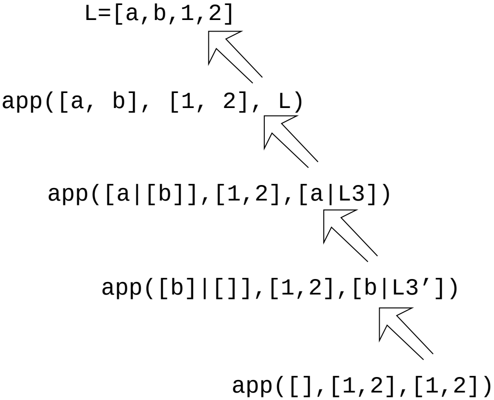
app([b|[]],[1,2],[b|L3′′])

(1)

app([],[1,2],[1,2])

succeed

Prolog Lists — Append. . .



`L=[a,b,1,2]`

`app([a, b], [1, 2], L)`

`app([a|[b]],[1,2],[a|L3])`

`app([b]|[]],[1,2],[b|L3′])`

`app([],[1,2],[1,2])`

```
?- L = [a | L3], L3 = [b | L3’], L3’ = [1,2].
      L = [a,b,1,2], L3 = [b,1,2], L3’ = [1,2]
```

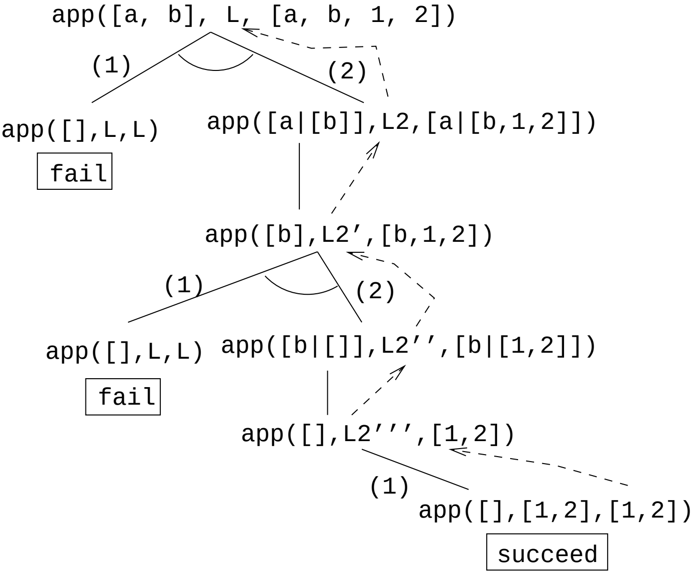
Prolog Lists — Using Append

1. `append([a,b], [1,2], L)`
    - What's the result of appending `[1,2]` onto `[a,b]`?
2. `append([a,b],[1,2],[a,b,1,2])`
    - Is `[a,b,1,2]` the result of appending `[1,2]` onto `[a,b]`?
3. `append([a,b], L, [a,b,1,2])`
    - What do we need to append onto `[a,b]` to make `[a,b,1,2]`?
    - What's the result of removing the prefix `[a,b]` from `[a,b,1,2]`?

Prolog Lists — Using Append. . .

4. `append(L, [1,2], [a,b,1,2])`
    - What do we need to append `[1,2]` onto to make `[a,b,1,2]`?
    - What's the result of removing the suffix `[1,2]` from `[a,b,1,2]`?
5. `append(L1, L2, [a,b,1,2])`
    - How can the list `[a,b,1,2]` be split into two lists L1 & L2?

Prolog Lists — Using Append. . .



app([a, b], L, [a, b, 1, 2])

(1)

(2)

app([],L,L)

fail

app([a|[b]],L2,[a|[b,1,2]])

app([b],L2',[b,1,2])

(1)

(2)

app([],L,L)

fail

app([b|[]],L2'',[b|[1,2]])

app([],L2''',[1,2])

(1)

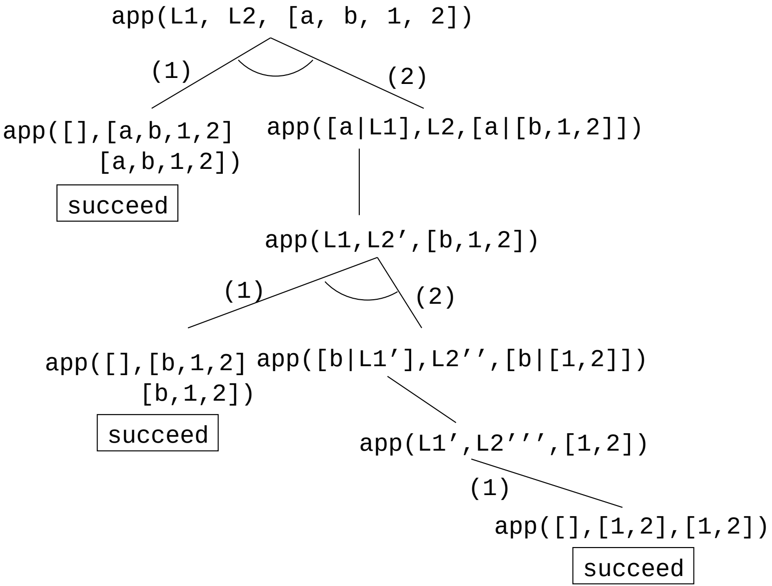
app([],[1,2],[1,2])

succeed

Prolog Lists — Using Append. . .

```
?- append(L1, L2, [a,b,c]).
      L1 = []
      L2 = [a,b,c] ;

      L1 = [a]
      L2 = [b,c] ;

      L1 = [a,b]
      L2 = [c] ;

      L1 = [a,b,c]
      L2 = [] ;

      no
```

Prolog Lists — Using Append. . .



`app(L1, L2, [a, b, 1, 2])`

(1)

(2)

`app([],[a,b,1,2]`
`[a,b,1,2])`

succeed

`app([a|L1],L2,[a|[b,1,2]])`

`app(L1,L2’,[b,1,2])`

(1)

(2)

`app([],[b,1,2]`
`[b,1,2])`

succeed

`app([b|L1’],L2’’,[b|[1,2]])`

`app(L1’,L2’’’,[1,2])`

(1)

`app([],[1,2],[1,2])`

succeed

Prolog Lists — Reusing Append

member	Can we split the list Y into two lists such that X is at the head of the second list?
adjacent	Can we split the list Z into two lists such that the two element X and Y are at the head of the second list?
last	Can we split the list Y into two lists such that the first list contains all the elements except the last one, and X is the sole member of the second list?

Prolog Lists — Reusing Append. . .

```
member(X, Y) :- append(_, [X|Z], Y).
   ?- member(x,[a,b,x,d]).

adjacent(X, Y, Z) :- append(_, [X,Y|Q], Z).
   ?- adjacent(x,y,[a,b,x,y,d]).

last(X, Y) :- append(_, [X], Y).
   ?- last(x, [a,b,x]).
```

Reversing a List

Prolog Lists — Reverse

- `reverse1` is known as *naive reverse*.
- `reverse1` is *quadratic* in the number of elements in the list.
- From *The Art of Prolog*, Sterling & Shapiro pp. 12-13, 203.
- Is the basis for computing LIPS (Logical Inferences Per Second), the performance measure for logic computers and programming languages. Reversing a 30 element list (using naive reverse) requires 496 reductions. A reduction is the basic computational step in logic programming.

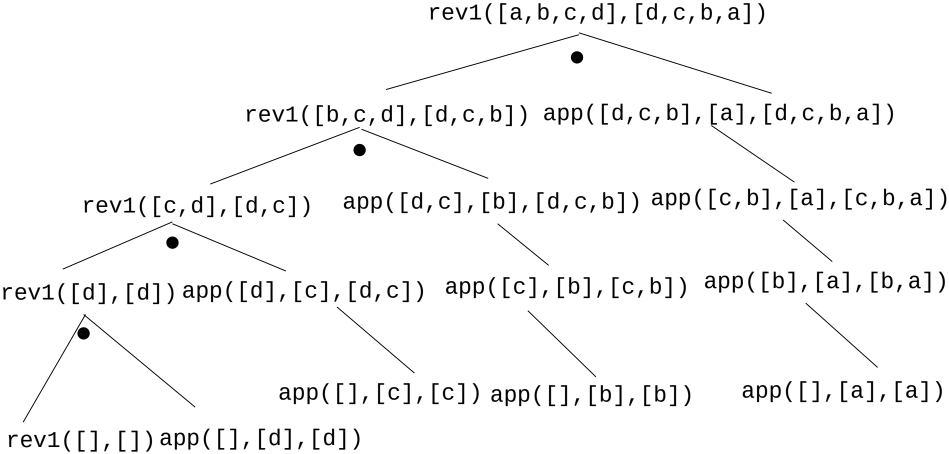
Prolog Lists — Reverse. . .

- `reverse1` works like this:
  1. Reverse the tail of the list.
  2. Append the head of the list to the reversed tail.
- `reverse2` is *linear* in the number of elements in the list.
- `reverse2` works like this:
  1. Use an accumulator pair `In` and `Out`
  2. `In` is initialized to the empty list.
  3. At each step we take one element (`X`) from the original list (`Z`) and add it to the beginning of the `In` list.
  4. When the original list (`Z`) is empty we instantiate the `Out` list to the result (the `In` list), and return this result up through the levels of recursion.

Prolog Lists — Reverse. . .

```
reverse1([], []).
reverse1([X|Q], Z) :-
   reverse1(Q, Y), append(Y, [X], Z).

reverse2(X, Y) :- reverse2(X, [], Y).
reverse2([X|Z], In, Out) :-
   reverse(Z, [X|In], Out).
reverse2([], Y, Y).
```

Reverse – Naive Reverse



rev1([a,b,c,d],[d,c,b,a])

rev1([b,c,d],[d,c,b]) app([d,c,b],[a],[d,c,b,a])

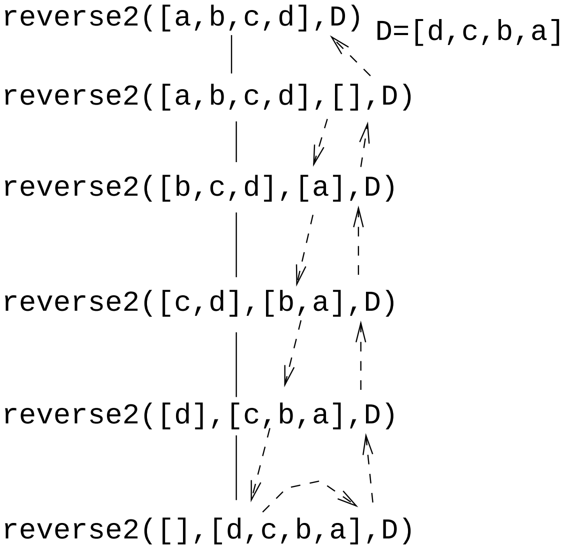
rev1([c,d],[d,c]) app([d,c],[b],[d,c,b]) app([c,b],[a],[c,b,a])

rev1([d],[d]) app([d],[c],[d,c]) app([c],[b],[c,b]) app([b],[a],[b,a])

app([],[c],[c]) app([],[b],[b]) app([],[a],[a])

rev1([],[]) app([],[d],[d])

Reverse – Smart Reverse



reverse2([a,b,c,d],D) D=[d,c,b,a]

reverse2([a,b,c,d],[],D)

reverse2([b,c,d],[a],D)

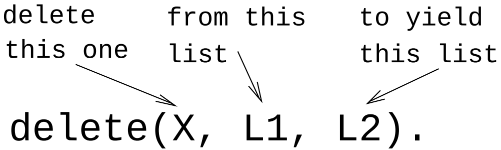
reverse2([c,d],[b,a],D)

reverse2([d],[c,b,a],D)

reverse2([],[d,c,b,a],D)

Delete

Prolog Lists — Delete. . .



delete this one

from this list

to yield this list

```
delete(X, L1, L2).
```

delete_one	Remove the first occurrence.
delete_all	Remove all occurrences.
delete_struct	Remove all occurrences from all levels of a list of lists.

Prolog Lists — Delete. . .

```
?- delete_one(x, [a, x, b, x], D).
      D = [a, b, x]
?- delete_all(x, [a, x, b, x], D).
      D = [a, b]
?- delete_all(x, [a, x, b, [c, x], x], D).
      D = [a, b, [c, x]]
?- delete_struct(x, [a, x, [c, x], v(x)], D).
      D = [a, b, [c], v(x)]
```

Prolog Lists — Delete. . .

delete_one

1. If X is the first element in the list then return the tail of the list.
2. Otherwise, look in the tail of the list for the first occurrence of X.

Prolog Lists — Delete. . .

delete_all

1. If the head of the list is X then remove it, and remove X from the tail of the list.
2. If X is *not* the head of the list then remove X from the tail of the list, and add the head to the resulting tail.
3. When we're trying to remove X from the empty list, just return the empty list.

Prolog Lists — Delete. . .

- Why do we test for the recursive boundary case (`delete_all(X,[],[])`) last? Well, it only happens once so we should perform the test as few times as possible.
- The reason that it works is that when the original list (the second argument) is `[]`, the first two rules of `delete_all` won't trigger. Why? Because, `[]` does not match `[H|T]`, that's why!

Prolog Lists — Delete. . .

delete_struct

1. The first rule is the same as the first rule in `delete_all`.
2. The second rule is also similar, only that we descend into the head of the list (in case it should be a list), as well as the tail.
3. The third rule is the catch-all for lists.
4. The last rule is the catch-all for non-lists. It states that all objects which are not lists (atoms, integers, structures) should remain unchanged.

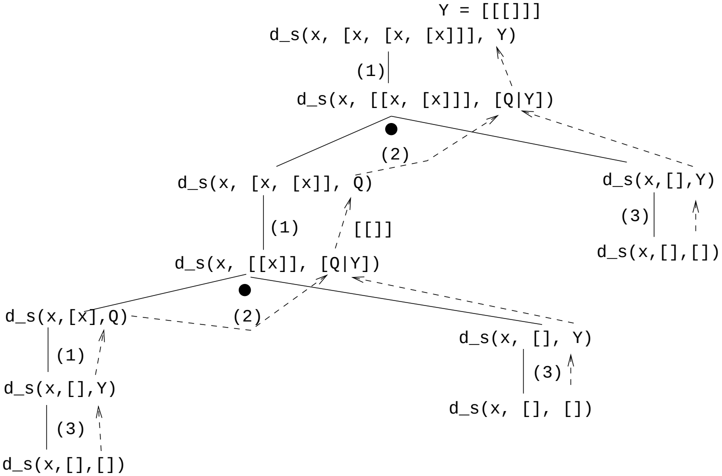
Prolog Lists — Delete. . .

```
delete_one(X,[X|Z],Z).
delete_one(X,[V|Z],[V|Y]) :-
      X \== V,
      delete_one(X,Z,Y).

delete_all(X,[X|Z],Y) :- delete_all(X,Z,Y).
delete_all(X,[V|Z],[V|Y]) :-
      X \== V,
      delete_all(X,Z,Y).
delete_all(X,[],[]).
```

Prolog Lists — Delete. . .

```
(1) delete_struct(X,[X|Z],Y) :-
      delete_struct(X, Z, Y).

(2) delete_struct(X,[V|Z],[Q|Y]):-
      X \== V,
      delete_struct(X, V, Q),
      delete_struct(X, Z, Y).

(3) delete_struct(X, [], []).
(4) delete_struct(X, Y, Y).
```

Prolog Lists — Delete. . .



Application: Sorting

Sorting – Naive Sort

```
permutation(X,[Z|V]) :-
   delete_one(Z,X,Y),
   permutation(Y,V).
permutation([],[]).

ordered([X]).
ordered([X,Y|Z]) :-
   X =< Y,
   ordered([Y|Z]).

naive_sort(X, Y) :-
   permutation(X, Y),
   ordered(Y).
```

Sorting – Naive Sort. . .

- This is an application of a Prolog cliche known as generate-and-test.

`naive_sort`

1. The `permutation` part of `naive_sort` generates one possible permutation of the input
2. The `ordered` predicate checks to see if this permutation is actually sorted.
3. If the list still isn't sorted, Prolog backtracks to the `permutation` goal to generate an new permutation, which is then checked by `ordered`, and so on.

Sorting – Naive Sort...

`permutation`

1. If the list is not empty we:
    1. Delete some element `Z` from the list
    2. Permute the remaining elements
    3. Add `Z` to the beginning of the list

   When we backtrack (ask `permutation` to generate a new permutation of the input list), `delete_one` will delete a different element from the list, and we will get a new permutation.
2. The permutation of an empty list is the empty list.

- Notice that, for efficiency reasons, the boundary case is put *after* the general case.

Sorting – Naive Sort. . .

`delete_one`	Removes the first occurrence of `X` (its first argument) from `V` (its second argument).

- Notice that when `delete_one` is called, its first argument (the element to be deleted), is an uninstantiated variable. So, rather than deleting a specific element, it will produce the elements from the input list (+ the remaining list of elements), one by one:

```
?- delete_one(X,[1,2,3,4],Y).
X = 1, Y = [2,3,4] ;
X = 2, Y = [1,3,4] ;
X = 3, Y = [1,2,4] ;
X = 4, Y = [1,2,3] ;
no.
```

Sorting – Naive Sort. . .

The proof tree in the next slide illustrates `permutation([1,2,3],V)`. The dashed boxes give variable values for each backtracking instance:

First instance: `delete_one` will select `X=1` and `Y=[2,3]`. `Y` will then be permuted into `Y'=[2,3]` and then (after having backtracked one step) `Y'=[3,2]`. In other words, we generate `[1,2,3]`, `[1,3,2]`.

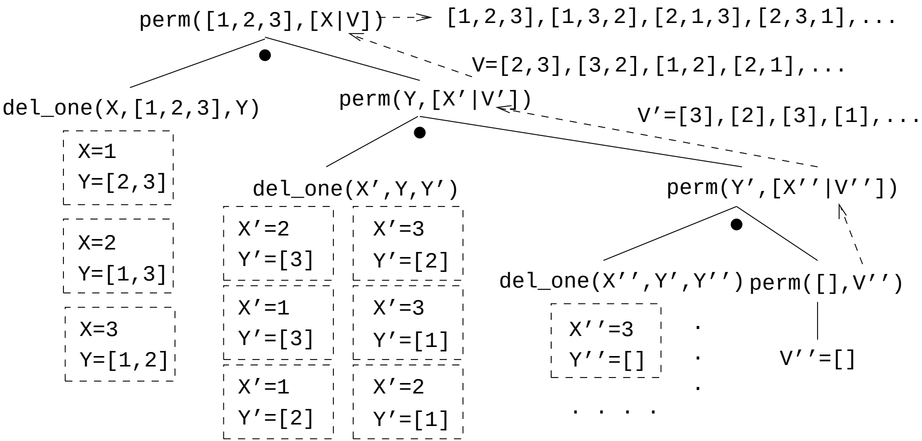
Second instance: We backtrack all the way back up the tree and select `X=2` and `Y=[1,3]`. `Y` will then be permuted into `Y'=[1,3]` and then `Y'=[3,2]`. In other words, we generate `[2,1,3]`, `[2,3,1]`.

Sorting – Naive Sort. . .

Third instance: Again, we backtrack all the way back up the tree and select `X=3` and `Y=[1,2]`. We generate `[3,1,2]`, `[3,2,1]`.

```
?- permutation([1,2,3],V).
V = [1,2,3] ;
V = [1,3,2] ;
V = [2,1,3] ;
V = [2,3,1] ;
V = [3,1,2] ;
V = [3,2,1] ;
no.
```

Permutations



perm([1,2,3],[X|V]) → [1,2,3],[1,3,2],[2,1,3],[2,3,1],...

V=[2,3],[3,2],[1,2],[2,1],...

del_one(X,[1,2,3],Y)

X=1
Y=[2,3]

X=2
Y=[1,3]

X=3
Y=[1,2]

perm(Y,[X'|V'])

V'=[3],[2],[3],[1],...

del_one(X',Y,Y')

X'=2
Y'=[3]

X'=3
Y'=[2]

X'=1
Y'=[3]

X'=3
Y'=[1]

X'=1
Y'=[2]

X'=2
Y'=[1]

perm(Y',[X''|V''])

del_one(X'',Y',Y'')

X''=3
Y''=[]

perm([],V'')

V''=[]

Sorting Strings

- Prolog strings are lists of ASCII codes.
- `"Maggie" = [77,97,103,103,105,101]`

```
aless(X,Y) :-
   name(X,X1), name(Y,Y1),
   alessx(X1,Y1).

alessx([],[_|_]).
alessx([X|_],[Y|_]) :- X < Y.
alessx([A|X],[A|Y]) :- alessx(X,Y).
```

Application: Mutant Animals

Mutant Animals

- From *Prolog by Example*, Coelho & Cotta.
- We're given a set of words (French animals, in our case).
- Find pairs of words where the ending of the first one is the same as the beginning of the second.
- Combine the words, so as to form new "mutations".

Mutant Animals. . .

1. Find two words, `Y` and `Z`.
2. Split the words into lists of characters. `name(atom, list)` does this.
3. Split `Y` into two sublists, `Y1` and `Y2`.
4. See if `Z` can be split into two sublists, such that the prefix is the same as the suffix of `Y` (`Y2`).
5. If all went well, combine the prefix of `Y` (`Y1`) with the suffix of `Z` (`Z2`), to create the mutant list `X`.
6. Use `name` to combine the string of characters into a new atom.

Mutant Animals...

```
mutate(M) :-
   animal(Y), animal(Z), Y \== Z,
   name(Y,Ny), name(Z,Nz),
   append(Y1,Y2,Ny), Y1 \==[],
   append(Y2, Z2, Nz), Y2 \== [],
   append(Y1,Nz,X), name(M,X).

animal(alligator).   /* crocodile*/
animal(tortue).      /* turtle   */
animal(caribou).     /* caribou  */
animal(ours).        /* bear     */
animal(cheval).      /* horse    */
animal(vache).       /* cow      */
animal(lapin).       /* rabbit   */
```

Mutant Animals. . .

```
?- mutate(X).
   X = alligatortue ;   /* alligator+ tortue */
   X = caribours ;      /* caribou  + ours */
   X = chevalligator ;  /* cheval   + alligator*/
   X = chevalapin ;     /* cheval   + lapin */
   X = vacheval         /* vache    + cheval */
```

Summary

Prolog So Far. . .

- Lists are nested *structures*
- Each list node is an object
  - with functor `.` (dot).
  - whose first argument is the head of the list
  - whose second argument is the tail of the list
- Lists can be split into head and tail using `[H|T]`.
- Prolog strings are lists of ASCII codes.
- `name(X,L)` splits the atom X into the string L (or vice versa).

CSc 372

Comparative Programming Languages

28 : Prolog — The Database

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

Manipulating the Database

- So far we have assumed that the Prolog database is static, i.e. that it is loaded once with the program and never changes thereafter.
- This is not necessarily true; we can add or remove facts and rules from the database at will.
- This is not necessarily good programming practice, but sometimes it is necessary and sometimes it makes for elegant programs.
- In a nutshell:
    1. Allows us to program with side effects.
    2. Justified under some circumstances.
    3. Often inefficient.

Operations on the Prolog Database

Assert

- `assert(X)` adds a clause to the database.
  Not defined in gprolog!
- `asserta(X)` adds a clause to the *beginning* of the database.
- `assertz(X)` adds a clause to the *end* of the database.
- `assert` always succeeds, and backtracking does not undo the assertion.

Assert. . .

- `assert` can be used in *machine learning* programs, program which learn new facts as they progress.
- In some Prolog implementations you have to specify whether a certain clause is dynamic (new clauses can be added to the database during execution) or static:

  ```
  :- dynamic(hanoi/5).
  ```

  This means that we can add and remove clauses with five arguments whose functor is `hanoi`.

Assert . . . – Example

- Write a program that learns the addresses of places in a city.
- This program assumes a Manhattan-style city layout:
  locations are given as the intersection of streets and avenues.

```
?- loc(whitehorse, Ave, St).
   Ave = 8, St = 11
?- loc(airport, Ave, St).
   -- this airport
   what avenue?  5.
   what street?  32.
   Ave = 5, St = 32
?- loc(airport, Ave, St).
   Ave = 5, St = 32
```

Assert . . . – Example

```
location(whitehorse, 8, 11).
location(microsoft, 8, 42).
location(condomeria, 8, 43).
location(plunket, 7, 32).

% Do we know the location of X?
loc(X, Ave, Str) :- location(X, Ave, Str), !.

% if not, learn it!
loc(X, Ave, Street) :-
   nonvar(X), var(Ave), var(Str),
   write('-- this '), write(X), nl,
   write('what avenue?  '), read(Ave),
   write('what street?  '), read(Street),
   assert(location(X, Ave, Str)).
```

Retract

- `retract(X)` removes the first clause that matches `X`.
- `assert` and `retract` behave differently on backtracking. When we backtrack through `assert` nothing happens. When we backtrack to `retract` Prolog continues searching the database trying to find another matching clause. If one is found it is removed.
- If the argument to `retract(clause(X))` contains some uninstantiated variables they will be instantiated.
- `retract(X)` fails when no matching clause can be found.

Retract…

- Backtracking does not undo the removal.

```
retractall(X) :-
   retract(X), fail.
retractall(X) :-
   retract((X :- Y))),
   fail.
retractall(_).
```

Clause

- `clause(X, Y)` finds all clauses in the database with head `X` and body `Y`.

```
append([], X, X).
append([A|B],C,[A|D]) :-
    append(B, C, D).
```

```
?- clause(append(X, Y, Z), T).
  X=[], Y=_3, Z=_3, Y=true ;
  X=[_4|_5], Y=_6, Z=[_4|_7],
     Y=append(_5, _6, _7) ;
  no
```

Clause. . .

- The goal `clause(X, Y)` instantiates `X` to the head of a goal (the left side of `:-`) and `Y` to the body.
- `X` can be just a variable (in which case it will match *all* the clauses in the database), a fully instantiated (*ground*) term, or a term which contains some uninstantiated variables.
- Note that a fact has a body `true`.

Clause...

List all the clauses whose head matches `X`*.*

```
list(X) :- clause(X, Y),
   print(X, Y),
   write('.'), nl, fail.
list(_).

print(X, true) :- !, write(X).
print(X, Y) :- write((X :- Y)).

?- list(append(X, Y, Z)).
   append([], _4, _4).
   append([_5|_6],_7,[_5|_8]) :-
         append(_6, _8, _8).
```

Clausal Representation of Data Structures

- Normally we represent a data structure using a combination of Prolog lists and structures.
- A graph can for example be represented as a list of edges, where each edge is represented by a binary structure:

  `[edge(a,b), edge(c,b), edge(a,d), edge(c,d)]`
- However, it is also possible to use *clauses* to represent data structures such as lists, trees, and graphs.
- It is usually not a good idea to do this, but sometimes it is useful, particularly when we are faced with a *static* data structure (one which does not change, or changes very little).

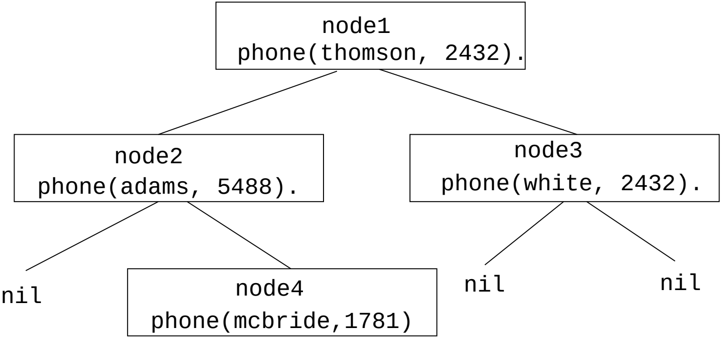
Clauses as Data Structures – Lists

```
list(c).
list(h).
list(r).
list(i).
list(s).

process_list :- list(X), process_item(X), fail.
process_list.
```

Clauses as Data Structures – Trees

```
t(node1, node2, phone(thompson, 2432), node3).
t(node2, nil, phone(adams, 5488), node4).
t(node3, nil, phone(white, 2432), nil).
t(node4, nil, phone(mcbride, 1781), nil).
```

Clauses as Data Structures – Trees. . .



node1

`phone(thomson, 2432).`

node2

`phone(adams, 5488).`

node3

`phone(white, 2432).`

`nil`

node4

`phone(mcbride,1781)`

`nil`

`nil`

Clauses as Data Structures – Trees. . .

```
inorder(nil).
inorder(Node) :-
   t(Node, Left, P, Right),
   inorder(Left),
   write(P), nl,
   inorder(Right).

?- inorder(node1).
   phone(adams,5488)
   phone(mcbride,1781)
   phone(thompson,2432)
   phone(white,2432)
```

Clausal Representation. . .

- In general it is a bad idea to represent data in this way.
- Inserting and removing data has to be done using `assert` and `retract`, which are fairly expensive operations.
- However, in Prolog implementations which support *clause indexing*, storing data in clauses gives us a way to access information *directly*, rather than through sequential search.
- The reason for this is that *indexing* uses hash tables to access clauses.

Switches

Switches

- From *Prolog by Example*, Coelho & Cotta.
- In some cases it is a good idea to use global data rather than passing it around as a parameter.
- Assume we want to be able to switch between short and long error messages. Instead of extending every clause by an extra parameter (clumsy and inefficient) we use a global switch.
- The first clause in `turnon` will fire if the switch is already turned on.
- The first clause in `turnoff` fails if `Switch` was already off.
- The first clause in `flip` fails if `Switch` was turned off, in which case the second clause fires and the switch is turned on.

Switches...

```
turnon(Switch) :-
   call(Switch), !.
turnon(Switch) :-
   assert(Switch).

turnoff(Switch) :-
   retract(Switch).
turnoff(_).

flip(Switch) :-
   retract(Switch), !.
flip(Switch) :-
   assert(Switch).
```

Switches...

```
turnon(terse_mess).
      .....
flip(terse_mess).

message(C) :-
   terse_mes, write ('Error!'), nl, !.
message(C) :-
   write ('We are sorry to...'),
   write ('error has occurred near the symbol '),
   write(C), write('.  Please accept our...'),
   nl, !.
```

Memoization

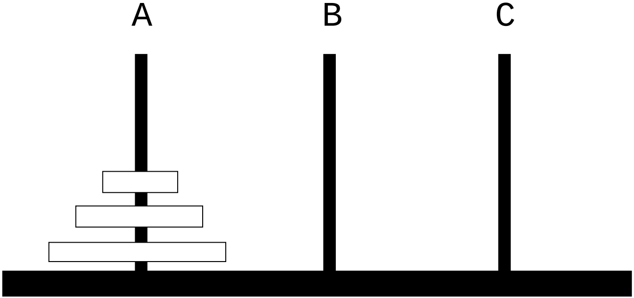
Memoization

- Many recursive program are extremely inefficient because they solve the same subproblem several times.
- In dynamic programming the idea is simply to store the results of a computation in a table, and when we try to solve the same problem again we retrieve the value from the table rather than computing the value once more.
- There is a variation of dynamic programming known as memoization.

Memoization – Towers of Hanoi

- I'm sure you've heard of the Towers of Hanoi problem. It is one first year computer science students are tortured with to no end.
- The problem is to move a number of disks from a peg A to a peg B, using a peg C as intermediate storage. Additionally, we are only allowed to put smaller disks onto larger disks.
- A recursive solution of the problem to move N disks from A to B is as follows:
  1. Move $N - 1$ disks from A to C.
  2. Move the remaining (largest) disk from A to B.
  3. Move the $N - 1$ disks from C to B.

Memoization – Towers of Hanoi…



Memoization – Towers of Hanoi. . .

```
:- op(100, xfx, to).

hanoi(1, A, B, C, [A to B]).
hanoi(N, A, B, C, Ms) :-
   N > 1,
   N1 is N-1,
   hanoi(N1, A, C, B, M1),
   hanoi(N1, C, B, A, M2),
   append(M1, [A to B|M2], Ms).

go(N, Moves) :-
   hanoi(N, a, b, c, Moves).
```

Memoization – Towers of Hanoi. . .

```
?- go(2,M).
   M = [a to c, a to b, c to b]

?- go(3,M).
   M = [a to b, a to c, b to c,
      a to b, c to a, c to b,
      a to b]

?- go(4,M).
   M = [a to c, a to b, c to b,
      a to c, b to a, b to c,
      a to c, a to b, c to b,
      c to a, b to a, c to b,
      a to c, a to b, c to b]
```

Memoization – Towers of Hanoi. . .

```
hanoi(1, A, B, C, [A to B]).
hanoi(N, A, B, C, Ms) :-
   N > 1, R is N-1,
   lemma(hanoi(R, A, C, B, M1)),
   hanoi(N1, C, B, A, M2),
   append(M1, [A to B|M2], Ms).

lemma(P) :- call(P),
         asserta((P :- !)).

go(N, Pegs, Moves) :-
   hanoi(N, A, B, C, Moves),
   Pegs=[A, B, C].
```

Memoization – Towers of Hanoi. . .

```
hanoi(1, _3, _5, _4, [_3 to _5]) :- !.
hanoi(2, _3, _4, _5,
     [_3 to _5, _3 to _4, _5 to _4]) :- !.
hanoi(3, _3, _5, _4,
     [_3 to _5, _3 to _4, _5 to _4,
     _3 to _5, _4 to _3, _4 to _5,
     _3 to _5]) :- !.
```

Example – Gensym

Example – Gensym

- From *Programming in Prolog*, Clocksin & Mellish.
- If we want to store data between different top-level queries, then using the database is our only option.
- In the following example we want to generate new atoms.
- In order to make this work, `gensym` has to store the number of atoms with a given prefix that it has generated so far. The clause `current_num(Root, Num)` is used for this purpose. There is one `current_num` clause for each kind of atom that we generate.

Example – Gensym. . .

```
gensym(Root, Atom) :-
   get_num(Root, Num),
   name(Root, Name1),
   int_name(Num, Name2),
   append(Name1, Name2, Name),
   name(Atom, Name).

get_num(Root, Num) :-
   retract(current_num(Root, Num1)),
   !, Num is Num1 + 1,
   asserta(current_num(Root, Num)).
get_num(Root, 1) :-
   asserta(current_num(Root, 1)).
```

Example – Gensym. . .

```
int_name(Int, List) :- int_name(Int, [], List).
int_name(I, Sofar, [C|Sofar]) :-
   I<10, !, C is I+48.
int_name(I, Sofar, List) :-
   Tophalf is I/10, Bothalf is I mod 10,
   C is Bothalf + 48,
   int_name(Tophalf, [C|Sofar], List).
```

```
?- gensym(chris, A).
   A = chris1
?- gensym(chris, A).
   A = chris2
?- gensym(chris, A).
   A = chris3
```

Readings and References

- Read Clocksin-Mellish, Chapter 6.

CSc 372

Comparative Programming Languages

29 : Prolog — Negation

Department of Computer Science
University of Arizona

collberg@gmail.com

The Cut

Cuts & Negation

The cut (`!`) is is ued to affect Prolog's backtracking. It can be used to

- reduce the search space (save time).
- tell Prolog that a goal is deterministic (has only one solution) (save space).
- construct a (weak form of) negation.
- construct `if_then_else` and `once` predicates.

Cuts & Negation

- The cut reduces the flexibility of clauses, and destroys their logical structure.
- Use cut as a last resort.
- Reordering clauses can sometimes achieve the desired effect, without the use of the cut.
- If you are convinced that you have to use a cut, try using `if_then_else`, `once`, or `not` instead.

The Cut

The Cut

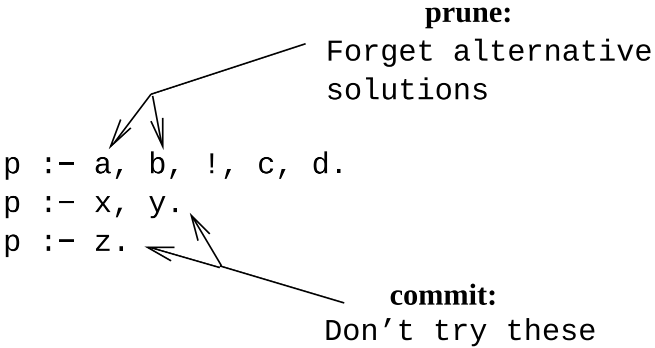
The cut succeeds and commits Prolog to all the choices made since the parent goal was called.

Cut does two things:

commit: Don't consider any later clauses for this goal.

prune: Throw away alternative solutions to the left of the cut.

The Cut



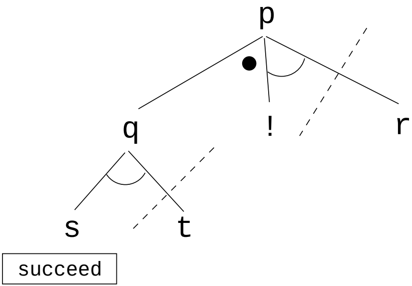
prune:
Forget alternative solutions

```
p :- a, b, !, c, d.
p :- x, y.
p :- z.
```

commit:
Don’t try these

The Cut

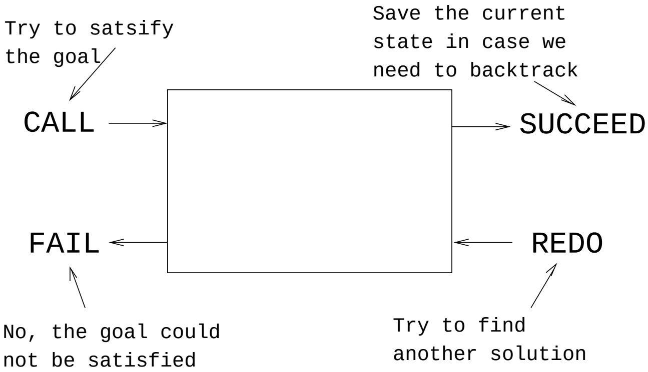
```
p :- q, !.
p :- r.
q :- s.
q :- t.
s.
```



The Boxflow Model

The Boxflow Model

The Boxflow Model



Try to satsify
the goal

Save the current
state in case we
need to backtrack

CALL

SUCCEED

FAIL

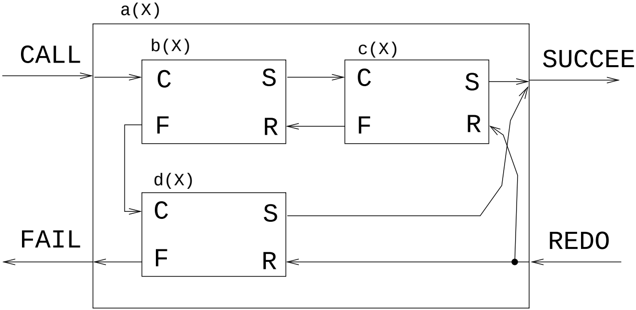
REDO

No, the goal could
not be satisfied

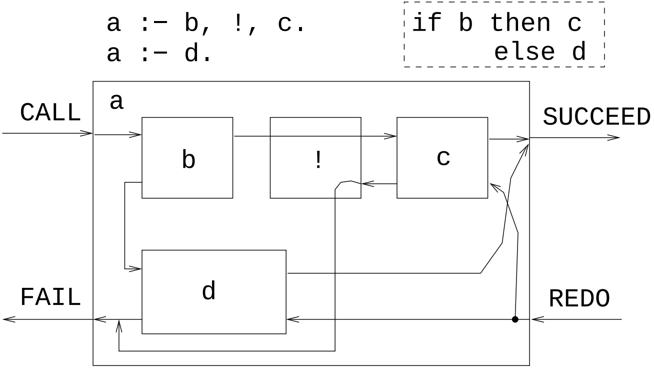
Try to find
another solution

The Boxflow Model

```
a(X) :- b(X), c(X).
a(X) :- d(X).
```



The Cut



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

```
if b then c
      else d
```

Classifying Cuts

Classifying Cuts

Classifying Cuts

grue No effect on logic, improves efficiency.

  green Prune away
  - irrelevant proofs
  - proofs which are bound to fail

  blue Prune away
  - proofs a smart Prolog implementation would not try, but a dumb one might.

red Remove unwanted logical solutions.

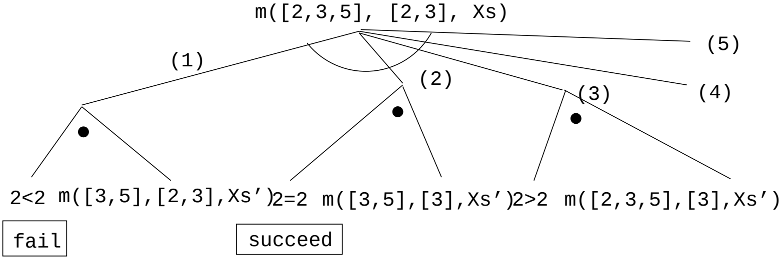
Green Cuts

Green Cuts – Merge

Produce an ordered list of integers from two ordered lists of integers.

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
   X < Y, merge(Xs, [Y|Ys], Zs).
merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
   X = Y, merge(Xs, Ys, Zs).
merge([X|Xs], [Y|Ys], [Y|Zs]) :-
   X > Y, merge([X|Xs], Ys, Zs).

merge(Xs, [], Xs).
merge([], Ys, Ys).

?- merge([1,4], [3,7], L).
   L = [1,3,4,7]
```

Green Cuts – Merge



m([2,3,5], [2,3], Xs)

(1)

(2)

(3)

(4)

(5)

2<2 m([3,5],[2,3],Xs′)

2=2 m([3,5],[3],Xs′)

2>2 m([2,3,5],[3],Xs′)

fail

succeed

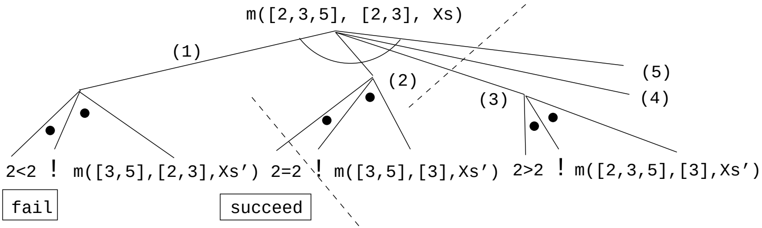
Green Cuts

- Still, there is no way for Prolog to know that the clauses are mutually exclusive, unless we tell it so. Therefore, Prolog must keep all choice-points (points to which Prolog might backtrack should there be a failure) around, which is a waste of space.
- If we insert cuts after each test we will tell Prolog that the procedure is deterministic, i.e. that once one test succeeds, there is no way any other test can succeed. Prolog therefore does not need to keep any choice-points around.

Green Cuts – Merge

```
merge([X|Xs], [Y|Ys], [X|Zs]) :-
   X < Y, !,
   merge(Xs, [Y|Ys], Zs).

merge([X|Xs], [Y|Ys], [X,Y|Zs]) :-
   X = Y, !,
   merge(Xs, Ys, Zs).

merge([X|Xs], [Y|Ys], [Y|Zs]) :-
   X > Y, !,
   merge([X|Xs], Ys, Zs).

merge(Xs, [], Xs) :- !.
merge([], Ys, Ys) :- !.
```

Green Cuts – Merge



Red Cuts

Red Cuts – Abs

```
abs1(X, X) :- X >= 0.
abs1(X, Y) :- Y is -X.
?- abs1(-6, X).
      X = 6 ;
?- abs1(6, X).
      X = 6 ;
      X = -6 ;

abs2(X, X) :- X >= 0, !.
abs2(X, Y) :- Y is -X.
?- abs2(-6, X).
      X = 6 ;
?- abs2(6, X).
      X = 6 ;
```

Red Cuts – Abs

```
abs3(X, X) :- X >= 0.
abs3(X, Y) :- X < 0,
            Y is -X.

?- abs3(-6, X).
      X = 6 ;
      no
?- abs3(6, X).
      X = 6 ;
      no
```

Red Cuts – Intersection

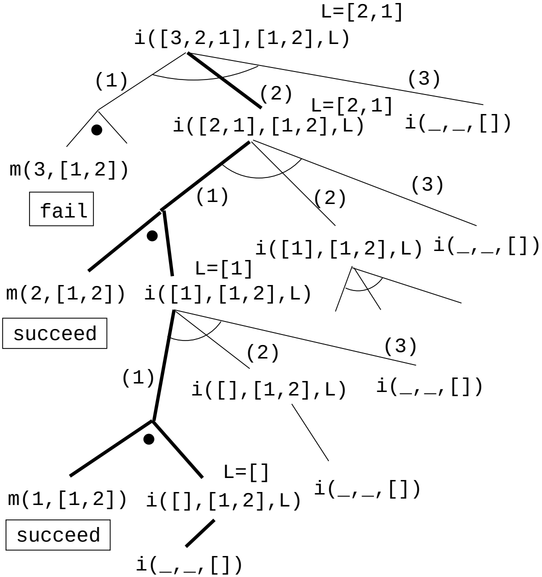
> *Find the intersection of two lists* `A` *&* `B`*, i.e. all elements of* `A` *which are also in* `B`*.*

```
intersect([H|T], L, [H|U]) :-
   member(H, L),
   intersect(T, L, U).
intersect([_|T], L, U) :-
   intersect(T, L, U).
intersect(_,_,[]).
```

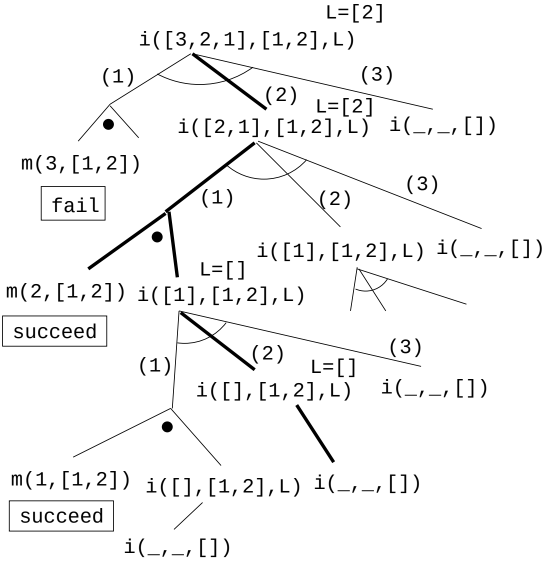
Red Cuts – Intersection

```
?- intersect([3,2,1],[1,2], L).
      L = [2,1] ;
      L = [2] ;
      L = [2] ;
      L = [1] ;
      L = [] ;
      L = [] ;
      L = [] ;
      L = [] ;
      no
```

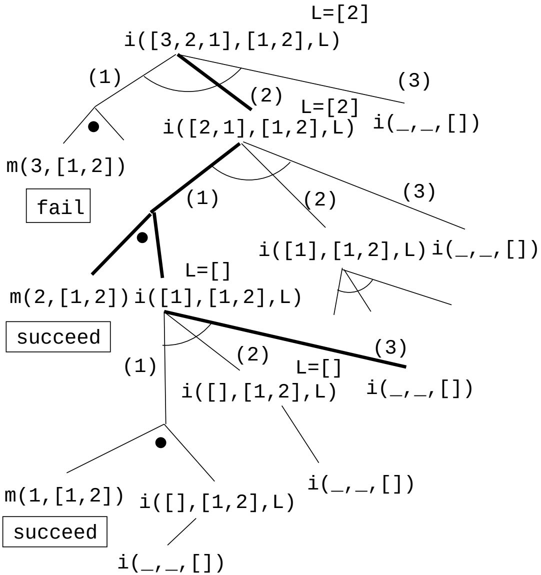
Red Cuts – Intersection



Red Cuts – Intersection



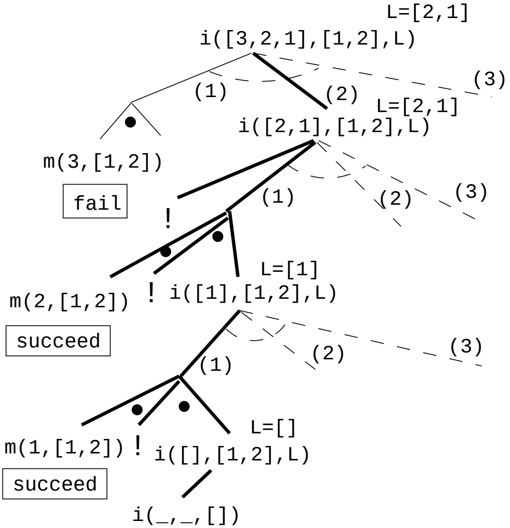
Red Cuts – Intersection



Red Cuts – Intersection

```
intersect([H|T], L, [H|U]) :-
   member(H, L),
   intersect(T, L, U).
intersect([_|T], L, U) :-
   intersect(T, L, U).
intersect(_,_,[]).

intersect1([H|T], L, [H|U]) :-
   member(H, L), !,
   intersect1(T, L, U).
intersect1([_|T], L, U) :-
   !, intersect1(T, L, U).
intersect1(_,_,[]).
```

Red Cuts – Intersection



Blue Cuts

Blue Cuts

First clause indexing will select the right clause in **constant** *time:*

```
clause(x(5), ...)  :- ...
clause(y(5), ...)  :- ...
clause(x(5, f), ...)  :- ...
?- clause(x(C, f),...).
```

First clause indexing will select the right clause in **linear** *time:*

```
clause(W, x(5), ...)  :- ...
clause(W, y(5), ...)  :- ...
clause(W, x(5, f), ...)  :- ...
?- clause(a, x(C, f),...).
```

Blue Cuts

```
capital(britain, london).
capital(sweden, stockholm).
capital(nz, wellington).
?- capital(sweden, X).
      X = stockholm
?- capital(X, stockholm).
      X = sweden

capital1(britain, london) :- !.
capital1(sweden, stockholm) :- !.
capital1(nz, wellington) :- !.
?- capital1(sweden, X).
      X = stockholm
?- capital1(X, stockholm).
      X = sweden
```

Once

Red Cuts – Once

```
member(H,[H|_]).
member(I, [_|T]) :- member(I, T).

?- member(1,[1,1]), write('x'), fail.
      xx

mem1(H,[H|_]) :- !.
mem1(I, [_|T]) :- mem1(I, T).
?- mem1(1, [1,1]), write('x'), fail.
      x
once(G) :- call(G), !.
one_mem(X, L) :- once(mem(X, L)).
?- one_mem(1,[1,1]), write('x'),fail.
      x
```

Red Cuts – Once

> *Red cuts prune away logical solutions. A clause with a red cut has no logical reading.*

```
?- member(X, [1,2]).
   X = 1 ;
   X = 2 ;
   no

?- one_mem(X, [1,2]).
   X = 1 ;
no
```

Cut & Fail & IF-THEN-ELSE

Red Cuts – Abs

```
abs2(X, X) :- X >= 0, !.
abs2(X, Y) :- Y is -X.

if_then_else(P,Q,R):-call(P),!,Q.
if_then_else(P,Q,R):-R.

abs4(X, Y) :- if_then_else(X >= 0,
                  Y=X, Y is -X).

?- abs4(-6, X).
      X = 6 ;
      no
?- abs4(6, X).
      X = 6 ;
      no
```

IF–THEN–ELSE

```
intersect([H|T], L, [H|U]) :-
   member(H, L), !, intersect(T, L, U).
intersect([_|T], L, U) :-
   !, intersect(T, L, U).
intersect(_,_,[]).
```

```
IF H ∈ L THEN
   compute the inters. of T and L,
   let H be in the resulting list.
ELSEIF the list \= [] THEN
   let the resulting list be the
   intersection of T and L.
ELSE
   let the resulting list be [].
ENDIF
```

IF–THEN–ELSE

```
if_then_else(P,Q,R) :- call(P), !, Q.
if_then_else(P,Q,R) :- R.

intersect2([X|T], L, W) :-
   if_then_else(member(X, L),
         (intersect2(T, L, U), W=[X|U]),
         if_then_else(T \= [],
               intersect2(T, L, W),
               W = [])).
```

Negation

Negation

Open vs. Closed World

How should we handle *negative information*?

Open World Assumption:

If a clause `P` *is not currently asserted then* `P` *is neither true nor false.*

Closed World Assumption:

If a clause `P` *is not currently asserted then the negation of* `P` *is currently asserted.*

Open vs. Closed World

```
striker(dahlin).
striker(thern).
striker(andersson).
```

Open World Assumption:

Dahlin, Thern, and Andersson are strikers, but there may be others we don't know about.

Closed World Assumption:

`X` *is a striker* if and only if `X` *is one of Dahlin, Thern, and Andersson.*

Negation in Prolog

- Prolog makes the closed world assumption.
- Anything that I do not know and cannot deduce is not true.
- Prolog's version of negation is *negation as failure*.
- `not(G)` means that `G` *is not satisfiable as a Prolog goal*.

```
(1)   not(G) :- call(G),!,fail.
(2)   not(G).
```

```
?- not(member(5, [1,3,5])).
no
?- not(member(5, [1,3,4])).
yes
```

Prolog Execution – Not

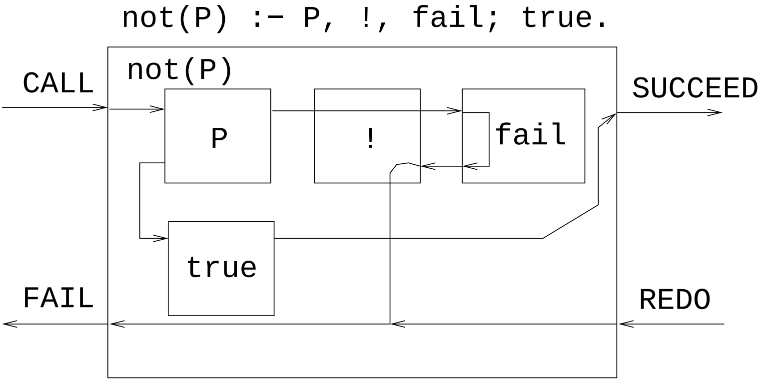
- Some Prolog implementations don't define `not` at all. We then have to give our own implementation:

```
(1)   not(G) :- call(G),!,fail.
(2)   not(G).
```

- Some implementations define `not` as
  - the operator `not`;
  - the operator \+;
  - the predicate `not(Goal)`.

`gprolog` uses \+.

Prolog Execution – Not



Negation Example – Disjoint

Do the lists `X` *&* `Y` **not** *have any elements in common?*

```
disjoint(X, Y) :-
      not(member(Z, X),
         member(Z, Y)).

?- disjoint([1,2],[3,2,4]).
      no

?- disjoint([1,2],[3,7,4]).
      yes
```

Prolog Negation Problems

```
man(john).  man(adam).
woman(sue). woman(eve).
married(adam, eve).

married(X) :- married(X, _).
married(X) :- married(_, X).
human(X) :- man(X).
human(X) :- woman(X).

% Who is not married?
?- not married(X).
   false

% Who is not dead?
?- not dead(X).
   true
```

Prolog Negation Problems

```
man(john).  man(adam).
woman(sue). woman(eve).
married(adam, eve).
married(X) :- married(X, _).
married(X) :- married(_, X).
human(X) :- man(X).
human(X) :- woman(X).

% Who is not married?
?- human(X), not married(X).
   X = john ; X = sue
% Who is not dead?
?- man(X), not dead(X).
   X = john ; X = adam ;
```

Prolog Negation Problems

- If `G` terminates then so does `not G`.
- If `G` does not terminate then `not G` may or may not terminate.

```
married(abraham, sarah).

married(X, Y) :- married(Y, X).

?- not married(abraham,sarah).
   false
?- not married(sarah,abraham).
   non-termination
```

Open World Assumption

We can program the *open world assumption*:

- A query is either *true*, *false*, or *unknown*.
- A false facts `F` has to be stated explicitly, using `false(F)`.
- If we can't prove that a statement is *true* or *false*, it's *unknown*.

```
% Philip is Charles' father.
father(philip, charles).

% Charles has no children.
false(father(charles, X)).
```

Open World Assumption

```
prove(P) :- call(P), write('** true'), nl,!.

prove(P) :- false(P), write('** false'), nl,!.

prove(P) :-
   not(P), not(false(P)),
   write('*** unknown'), nl, !.
```

Open World Assumption

```
father(philip, charles).
false(father(charles, X)).

% Is Philip the father of ann?
?- prove(father(philip, ann)).
   ** unknown

% Does Philip have any children?
?- prove(father(philip, X)).
   ** true
   X = charles

% Is Charles the father of Mary?
?- prove(father(charles, mary)).
   ** false
```

CSc 372

Comparative Programming Languages

30 : Prolog — Techniques

Department of Computer Science
University of Arizona

collberg@gmail.com

Generate & Test – Integer Division

Generate & Test

A generate-and-test procedure has two parts:

1. A generator which can generate a number of possible solutions.
2. A tester which succeeds iff the generated result is an acceptable solution.

When the tester fails, the generator will backtrack and generate a new possible solution.

Generate & Test – Division

- We can define integer arithmetic (inefficiently) in Prolog:

```
% Integer generator.
is_int(0).
is_int(X) :- is_int(Y), X is Y+1.

% Result = N1 / N2.
divide(N1, N2, Result) :-
   is_int(Result),
   P1 is Result*N2,
   P2 is (Result+1)*N2,
   P1 =< N1, P2 > N1, !.

| ?- divide(6,2,R).
      R = 3
```

Generate & Test – Division. . .

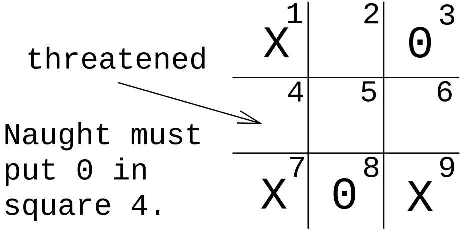
```
is_int(0).
is_int(X) :- is_int(Y), X is Y+1.
divide(N1, N2, Result) :-
   is_int(Result),
   P1 is Result*N2, P2 is (Result+1)*N2,
   P1 =< N1, P2 > N1, !.
```

`divide(6,2,R) --- N1=6, N2=2`				
`Res`	`P1`	`P2`	`P1 =< N1`	`P2 > N1`
0	0	2	True	False
1	2	4	True	False
2	4	6	True	False
3	6	12	True	True

Generate & Test – Tic-Tac-Toe

Generate & Test – Tic-Tac-Toe

- This is a part of a program to play Tic-Tac-Toe (Naughts and Crosses).
- Two players take turns to put down `X` and `O` on a 3x3 board. Whoever gets a line of 3 (horizontal, vertical, or diagonal) markers has won.



Generate & Test – Tic-Tac-Toe. . .

- We'll look at the predicate `forced_move` which answers the question:
  - Am I (the naught-person) forced to put a marker at a particular position?
- The program tries to find a line with two crosses.
- It only makes sense to find one forced move, hence the cut.

Generate & Test – Tic-Tac-Toe. . .

- `aline(L)` is a generator – it generates all possible lines(L).
- `threatening(L,B,Sq)` is a tester – it succeeds if `Sq` is a threatened square in line `L` of board `B`.

```
forced_move(Board, Sq) :-
   aline(Line),
   threatening(Line, Board, Sq), !.
```

```
?- forced_move(b(x,_,o,_,_,_,x,o,x),4).
   yes
```

```
aline([1,2,3]).  aline([4,5,6]).  aline([7,8,9]).
aline([1,4,7]).  aline([2,5,8]).  aline([3,6,9]).
aline([1,5,9]).  aline([3,5,7]).
```

Gen. & Test – Tic-Tac-Toe. . .

- `threatening` succeeds if it finds a line with two crosses and one empty square.

```
threatening([X,Y,Z],B,X) :-
   empty(X,B), cross(Y,B), cross(Z,B).
threatening([X,Y,Z],B,Y) :-
   cross(X,B), empty(Y,B), cross(Z,B).
threatening([X,Y,Z],B,Z) :-
   cross(X,B), cross(Y,B), empty(Z,B).
```

Gen. & Test – Tic-Tac-Toe. . .

- A square is empty if it is an uninstantiated variable.
- `arg(N,S,V)` returns the `N`:th element of a structure `S`.

```
empty(Sq, Board) :-
   arg(Sq,Board,Val), var(Val).
cross(Sq, Board) :-
   arg(Sq,Board,Val), nonvar(Val), Val=x.
naught(Sq, Board) :-
   arg(Sq,Board,Val), nonvar(Val), Val=o.
```

Arbitrage

Generate & Test – Arbitrage

From the Online Webster's:

arbitrage *simultaneous purchase and sale of the same or equivalent security in order to profit from price discrepancies*

```
?- arbitrage.
   dollar dmark yen 1.03751
   yen dollar dmark 1.03751
   dmark yen dollar 1.03751
```

Generate & Test – Arbitrage...

```
arbitrage :-
   profit3(From, Via, To, Profit), % Gen
   Profit > 1.03, % Test
   write(From), write(' '),
   write(Via), write(' '),
   write(To), write(' '),
   write(Profit), nl, fail.
arbitrage.

% Find three currencies, and the profit:
profit3(From, Via, To, Profit) :-
   best_rate(From, Via, P1, R1),
   best_rate(Via, To, P2, R2),
   best_rate(To, From, P3, R3),
   Profit is R1 * R2 * R3.
```

```
exchange(pound, dollar, london, 1.550).
exchange(pound, dollar, new_york, 1.555).
exchange(pound, dollar, tokyo, 1.559).
exchange(pound, yen, london, 153.97).
exchange(pound, yen, new_york, 154.05).
exchange(pound, yen, tokyo, 154.3).
exchange(pound, dmark, london, 2.4075).
exchange(pound, dmark, new_york, 2.44).
exchange(pound, dmark, tokyo, 2.408).
exchange(dollar, yen, london, 98.3).
exchange(dollar, yen, new_york, 98.35).
exchange(dollar, yen, tokyo, 98.25).
exchange(dollar, dmark, london, 1.537).
exchange(dollar, dmark, new_york, 1.58).
exchange(dollar, dmark, tokyo, 1.57).
exchange(yen, dmark, london, 0.015635).
exchange(yen, dmark, new_york, 0.0155).
exchange(yen, dmark, tokyo, 0.0158).
```

Generate & Test – Arbitrage...

```
% We can convert back and forth
% between currencies:
rate(From, To, P, R) :-
   exchange(From, To, P, R).
rate(From, To, P, R) :-
   exchange(To, From, P, S), R is 1/S.

% Find the best place to convert
% between currencies From & To:
best_rate(From, To, Place,Rate):-
   rate(From, To, Place, Rate),
   not((rate(From, To, P1, R1), R1>Rate)).
```

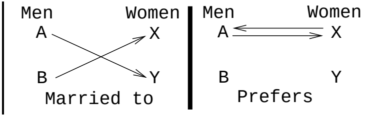
Stable Marriages

Stable Marriages

- Suppose there are N men and N women who want to get married to each other.
- Each man (woman) has a list of all the women (men) in his (her) preferred order. The problem is to find a set of marriages that is stable.

A set of marriages is `unstable` if two people who are not married both prefer each other to their spouses. If A and B are men and X and Y women, the pair of marriages $A - Y$ and $B - X$ is unstable if

- A prefers X to Y, and
- X prefers A to B.



Stable Marriages – Example

Person	Sex	1st choice	2nd choice	3rd choice
Avraham	M	Chana	Ruth	Zvia
Binyamin	M	Zvia	Chana	Ruth
Chaim	M	Chana	Ruth	Zvia
Zvia	F	Binyamin	Avraham	Chaim
Chana	F	Avraham	Chaim	Binyamin
Ruth	F	Avraham	Binyamin	Chaim

- Chaim-Ruth, Binyamin-Zvia, Avraham-Chana is stable.
- Chaim-Chana, Binyamin-Ruth, Avraham-Zvia is unstable, since Binyamin prefers Zvia over Ruth and Zvia prefers Binyamin over Avraham.

Stable Marriages. . .

- Write a program which takes a set of people and their preferences as input, and produces a set of stable marriages as output.

Input Format:

```
prefer(avraham, man,
         [chana,tamar,zvia,ruth,sarah]).

men([avraham,binyamin,chaim,david,elazar]).
women([zvia, chana, ruth, sarah, tamar]).
```

- The first rule, says that `avraham` is a man and that he prefers `chana` to `tamar`, `tamar` to `zvia`, `zvia` to `ruth`, and `ruth` to `sarah`.

```
prefer(avraham, man, [chana, tamar, zvia, ruth, sarah]).
prefer(binyamin, man, [zvia, chana, ruth, sarah, tamar]).
prefer(chaim, man, [chana, ruth, tamar, sarah, zvia]).
prefer(david, man, [zvia, ruth, chana, sarah, tamar]).
prefer(elazar, man, [tamar, ruth, chana, zvia, sarah]).
prefer(zvia, woman, [elazar, avraham, david, binyamin, chaim]).
prefer(chana, woman, [david, elazar, binyamin, avraham, chaim]).
prefer(ruth, woman, [avraham, david, binyamin, chaim, elazar]).
prefer(sarah, woman, [chaim, binyamin, david, avraham, elazar]).
prefer(tamar, woman, [david, binyamin, chaim, elazar, avraham]).
```

Stable Marriages...

- `gen` generates all possible sets of marriages, `unstable` tests if they are stable.

```
go :-
   men(ML), women(WL),
   gen(ML, WL, [], L), \+unstable(L)),
   show(L), fail.
go.
```

```
?- men(ML), women(WL), gen(ML,WL,[],L).
   L = [m(elazar,tamar),m(david,sarah),
      m(chaim,ruth),m(binyamin,chana),
      m(avraham,zvia)] ?  ;
         .........
```

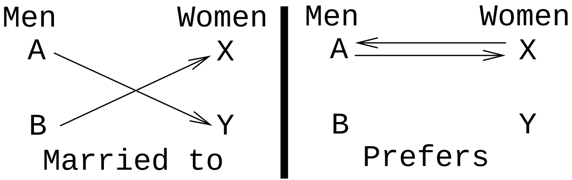
Stable Marriages — Generate

```
gen([A|M1], W, In, Out) :-
   delete(B, W, W1),
   gen(M1, W1, [m(A,B)|In], Out).
gen([],[],L,L).

delete(A, [A|L], L).
delete(A, [X|L], [X|L1]) :-
   delete(A, L, L1).
```

Stable Marriages — Test

```
% A prefers B to C.
pref(A, B, C) :-
   prefer(A, _, L),
   append(_, [B|S], L), !,
   member(C, S), !.

unstable(L) :-
   append(_, [A|R], L),
   member(B, R),
   (is_unstable(A,B);
      is_unstable(B,A)).

is_unstable(m(A,Y), m(B,X)) :-
   pref(A, X, Y),
   pref(X, A, B).
```

Stable Marriages...



Men

Women

A

X

B

Y

Married to

Men

Women

A

X

B

Y

Prefers

Bedtime Story

Puzzles – Bedtime Story

> *"Helder, a poor scientist, was in love with the daughter of an admiral. One day, a general captured the girl. Helder rode to the general's barrack and killed the general. The girl was grateful and fell in love with Helder. The admiral was so happy to have his daughter back he gave Helder half of all his boats."*

- "Who is the father of the girl?"
- "Who is rich?"
- "Who loves who?"
- "Who is poor?"
- "Who captured who?"
- "Who killed who?"

Puzzles – Bedtime Story. . .

```
:- op(500, xfy, 'is_').
:- op(500, yfx, 'loves').
:- op(500, yfx, 'kills').
:- op(500, yfx, 'to').
:- op(500, yfx, 'captures').
:- op(500, yfx, 'rides_to').
:- op(500, yfx, 'gives').
:- op(500, yfx, 'is_father_of').
:- op(800, yfx, 'and').

X and Y :- X, Y.
```

Puzzles – Bedtime Story. . .

```
helder is_ poor.
helder is_ scientist.
admiral is_ happy.
admiral is_father_of girl.
helder loves girl.
girl loves helder.
general captures girl.
helder kills general.
admiral gives half_boats to helder.
```

Puzzles – Bedtime Story…

```
% Who loves who?
?- Z loves Y, write(Z), write(' loves '),
      write(Y), nl, fail.
   helder loves girl
   girl loves helder

% Who captures who?
?- Z captures Y.
   Z = general
   Y = girl
```

Puzzles – Bedtime Story…

```
% Who kills who?
?- Z kills Y.
   Z = helder
   Y = general

% Who loves who's daughter?
?- Z loves G and F is_father_of G.
   Z = helder
   G = girl
   F = admiral
```

Puzzles – Trees

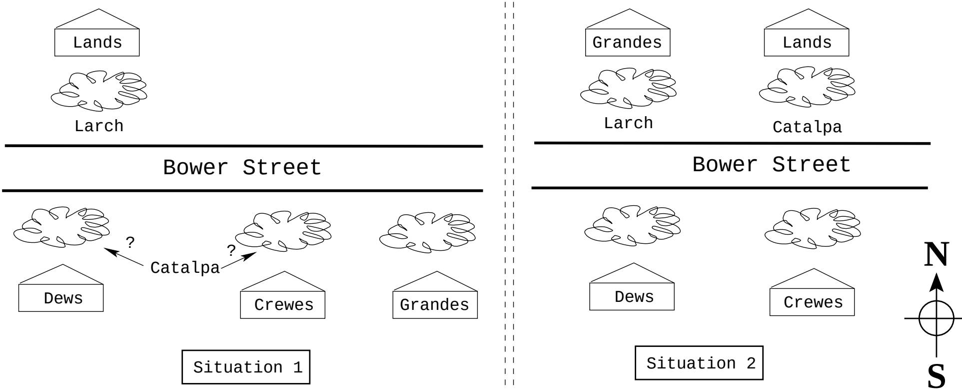
Puzzles – Trees

- The Crewes, Dews, Grandes, and Lands of Bower Street each have a front-yard tree: Catalpa, Dogwood, Gingko, Larch.
- The Grandes' tree and the Catalpa are on the same side of the street.
- The Crewes live across the street from the Larch.
- The Larch is across the street from the Dews' house.
- No tree starts with the same letter as its owner's name.
- Who owns which tree?

Puzzles – Trees

```
| ?- solve.
   Grandes owns the Larch
   Crewes owns the Dogwood
   Dews owns the Ginko
   Lands owns the Catalpa
```

Puzzles – Trees. . .



Puzzles – Trees. . .

```
% Let's assume that the Larch is on the
% north side of the street.
northside('Larch').

% The Crewes live across the street from
% the Larch.  The Larch is across the
% street from the Dews' house.
southside('Crewes').
southside('Dews').

% The Grandes' tree and the 'Catalpa'
% are on the same side of the street.
northside('Catalpa') :-
   northside('Grandes').
```

Puzzles – Trees. . .

```
% If Grandes have a 'Larch', then they
% must live on the north side.
northside('Grandes') :-
   have('Grandes', 'Larch').

% Grandes have a 'Larch', if noone
% else does.
have('Grandes','Larch') :-
   not_own('Crewes','Larch'),
   not_own('Dews','Larch'),
   not_own('Lands','Larch')
```

Puzzles – Trees. . .

```
% then the Dews' and Crews' will be
% on the south side.  Also, if the
% Catalpa is on the north the Dogwood
% and Ginko must both be on the south
% side (since each house has one tree).
southside('Dogwood') :-
   northside('Larch'),
   northside('Catalpa').
southside('Ginko') :-
   northside('Larch'),
   northside('Catalpa').
```

Puzzles – Trees. . .

```
% Are you a tree or a plant?
person(X) :- member(X,
   ['Grandes','Crewes','Dews','Lands']).
tree(X) :- member(X,
   ['Catalpa','Ginko','Dogwood','Larch']).

% No tree starts with the same letter as
% its owner's name.
not_own(X,Y) :-
   name(X, [A|_]), name(Y,[A|_]).

% The Grandes' tree and the 'Catalpa'
% are on the same side of the street.
not_own('Grandes','Catalpa').
```

Puzzles – Trees. . .

```
% Only a person can own a tree.
not_own(X,Y) :- person(X), person(Y).
not_own(X,Y) :- tree(X), tree(Y).

% A person can only own a tree that's on
% the same side of the street as
% themselves.
not_own(X,Y) :- northside(X),southside(Y).
not_own(X,Y) :- southside(X),northside(Y).
```

Puzzles – Trees. . .

```
% You can't own what someone else owns.
not_own('Crewes', X) :- owns('Dews', X).
not_own('Lands', X) :- owns('Crewes', X).
not_own('Lands', X) :- owns('Dews',X).

owns(X,Y) :-
   person(X), tree(Y),
   not(not_own(X,Y)).

solve :-
   owns(Person,Tree),
   write(Person), write(' owns the '),
   write(Tree),nl,fail.
solve.
```

Logic Arithmetic

Arithmetic In Logic

- Arithmetic in Prolog is just like arithmetic in imperative languages. We can't do $\boxed{\texttt{25 is X + Y}}$ and hope to get `X` and `Y` instantiated to every pair of numbers that sum to 25.
- There are cases when we need the power of logic arithmetic, rather than the efficient built-in operators. That is no problem, we can always define the logic arithmetic predicates ourselves.
- For example, how do we split a number into the two parts Note that this is similar to splitting a list using `append`.

Arithmetic In Logic. . .

- We can always write our own **logic** arithmetic predicates.

```
% Represent S as the sum of 2 numbers.
% minus(S, D1, D2) -- S − D₁ = D₂
minus(S, S, 0).
minus(S, D1, D2) :-     % Note that
   S > 0, S1 is S-1,    % S must be
   minus(S1, D1, D3),   % instantiated.
   D2 is D3 + 1.

?- minus(3, X, Y).
   X = 3, Y = 0 ;
   X = 2, Y = 1 ;
   X = 1, Y = 2 ;
   X = 0, Y = 3
```

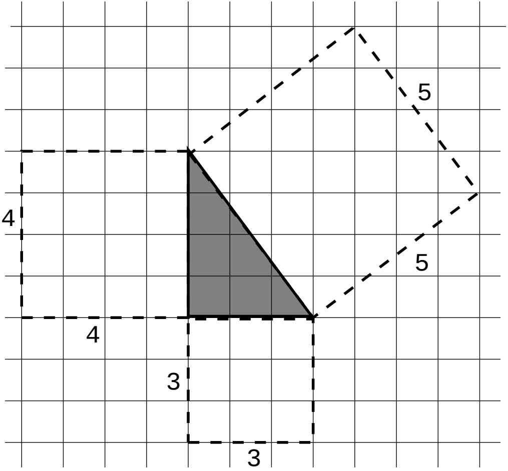
Arithmetic In Logic…

- The `minus` predicate splits `S` into `D1 + D2`. Why does it work? Well, look at this:

$$\begin{aligned} S1 &= S - 1 \ \texttt{first line} \\ D3 &= S1 - D1 \ \texttt{second line} \\ D2 &= D3 + 1 \ \texttt{third line} \\ S &= S1 + 1 \\ &= (D3 + D1) + 1 \\ &= ((D2 - 1) + D1) + 1 \\ &= D2 + D1 \end{aligned}$$

- Note that the `minus` predicate require the first argument to be instantiated, but not the second and third. `minus`, below, is a lot like `append`.

Pythagorean Triples



Pythagorean Triples. . .

```
?- pythag(X, Y, Z).
   X = 4, Y = 3, Z = 5 ;
   X = 3, Y = 4, Z = 5 ;
   X = 8, Y = 6, Z = 10 ;
   X = 6, Y = 8, Z = 10 ;
   X = 12, Y = 5, Z = 13 ;
   X = 5, Y = 12, Z = 13 ;
   X = 12, Y = 9, Z = 15
```

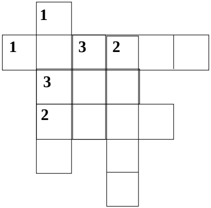
Pythagorean Triples. . .

- `is_int` is used to generate a sequence of numbers.
- `int_triple` splits the generated integer `S` into the sum of three integer `X, Y, Z`.
- In other words, first we check all triples that sum to 1 to see if any of them are pythagorean triples, then all triples that sum to 2, etc. This obviously will eventually check "all" triples. It also will make sure that we get them "in order", with the smallest triples first.

Pythagorean Triples...

```
% Generate a sequence of numbers.
is_int(0).
is_int(X) :- is_int(Y), X is Y+1.

pythag(X, Y, Z) :-
      int_triple(X, Y, Z),
      Z*Z =:= X*X + Y*Y.

% Generate integer triples:  S=X+Y+Z.
int_triple(X, Y, Z) :-
      is_int(S),
      minus(S, X, S1), X > 0,
      minus(S1, Y, Z), Y > 0, Y > 0.
```

Exercise: Crossword Puzzle



Across

1. The Fifth Element.
2. Mumintroll mum.
3. Beer.

Down

1. Kills at chess.
2. Best drummer. Ever.
3. Electric Light Orchestra.

Write a program that solves the crossword puzzle above, assuming this database of words:

```
word(leeloo). word(death). word(ale).
word(tove). word(levon). word(elo).
```

Exercise: Crossword Puzzle

1. Now, assume that you have a much bigger database of words.
2. How would you organize the database for much faster searching?
3. How would you rewrite your code to make use of the new database structure?

CSc 372

Comparative Programming Languages

31 : Prolog — Exercises

Department of Computer Science
University of Arizona

collberg@gmail.com

Problem I

Write a procedure `islist` which succeeds if its argument is a list, and fails otherwise.

Problem II

Write a procedure `alter` which changes English sentences according to rules given in the database.
Example:

```
change(you, i).
change(are, [am, not]).
change(french, german).
change(do, no).
?- alter([do,you,know,french],X).
   X = [no,i,know,german]
?- alter([you,are,a,computer],X).
   X = [i,[am,not],a,computer]
```

Problem III

Write a list subtraction procedure.
Example:

```
?- sub([1,2,4,6,8], [2,6], L).
   L=[1,4,8].
```

Problem IV

Write a procedure `pick` which returns the first `N` elements of a given list.
Example:

```
?- pick([1,2,4,6,8], 3, L).
   L=[1,2,4].
```

Problem V

Write a procedure `alt` which produces every other element in a list. Example:

```
?- alt([1,2,3,4,5,6], A).
   A = [1,3,5]
```

Problem VI

Write a procedure `del` which removes duplicate elements from a list.

Example:

```
?- del([a,c,x,a,g,c,d,a], A).
   A = [a,c,x,g,d]
```

Problem VII

Write a procedure `tolower` which converts an atom containing upper case characters to the corresponding atom with only lower case characters.
Example:

```
?- tolower('hEj_HoPp3', A).
   A = hej_hopp3
```

Problem VIII

Write a procedure `max3` which produces the largest of three integers.
Example:

```
?- max3(3,5,1,X).
   X = 5
```

Problem IX

Write a procedure `double` which multiplies each element in a list of numbers by 2.
Example:

```
?- double([1,5,3,9,2], A).
   A = [2,10,6,18,4]
```

Problem X

Write a procedure `ave` which computes the average of a list of numbers.

Example:

```
?- ave([1,5,3,9,2], A).
   A = 4
```

Problem XI

Write a procedure `sum` which produces the sum of the integers up to and including its first argument.
Example:

```
?- sum(5, S).
   S = 15
```

Problem XII

Suppose our database contains facts of the form

```
person_age(Name, Age).
person_sex(Name, Sex).
```

where `Sex` is either `male` or `female`. Write a procedure `combine` which extends the database with additional facts of the form

```
person_full(Name, Age, Sex).
```

The procedure should produce one such fact for each person who has both an age record and a sex record.

Problem XII…

Example: Given the following database

```
person_age(chris, 25).  % Yeah, right...
person_sex(chris, male).
person_age(louise, 8).
person_sex(louise, female).
```

`combine` should produce these additional facts:

```
person_full(chris, 25, male).
person_full(louise, 8, female).
```

Problem XIII

Write a Prolog procedure which reverses the order of Johns children in the database. For example, given the following database

```
child(mary, john).
child(jane, john).
child(bill, john).
```

the goal `?- reversefacts.` should change it to

```
child(bill, john).
child(jane, john).
child(mary, john).
```

Problem XIV

Write a Prolog procedure to assemble a list of someone's children from the facts in the database. The database should remain unchanged.
Example:

```
child(mary, john).
child(jane, john).
child(bill, john).

?- assemble(john, L).
   L = [mary, jane, bill]
```

Problem XV

Write down the *all* results (including variable bindings) of the following query:

```
?- append([], [1, 2|B], C),
   append([3,4], [5], B).
```

Problem XVI

Write down the *all* results (including variable bindings) of the following query:

```
?- bagof(X, Y^append(X, Y, [1,2,3,4]), Xs).
```

Problem XVII

Write down the *all* results (including variable bindings) of the following query:

```
?- L=[1,2], member(X, L), delete(X, Y, L).
```

Problem XVIII

Write down the *all* results (including variable bindings) of the following query:

```
?- member(X, [a,b,c]), member(Y, [a,b,c]), !, X \= Y.
```

Problem XIX

Given the following Prolog database

```
balance(john, 100).
balance(sue, 200).
balance(mary, 100).
balance(paul, 500).
```

list *all* the results of these Prolog queries:

1. `?- bagof(Name, balance(Name, Amount), Names).`
2. `?- bagof(Name, Amount^balance(Name, Amount), Names).`
3. `?- bagof(Name, Name^balance(Name, Amount), Names).`

Problem XX

Describe (in English) what the following predicate does:

```
% Both arguments to bbb are lists.
bbb([], []).
bbb(A, [X|F]) :- append(F, [X], A).
```

Problem XXI

Given the following program

```
a(1,2).
a(3,5).
a(R, S) :- b(R, S), b(S, R).

b(1,3).
b(2,3).
b(3, T) :- b(2, T), b(1, T).
```

list the first answer to this query:

```
?- a(X, Y), b(X, Y)
```

Will there be more than one answer?

Problem XXII

Given the following definitions:

```
f(1, one).
f(s(1), two).
f(s(s(1)), three).
f(s(s(s(X))), N) :- f(X, N).
```

what are the results of these queries? If there is more than one possible answer, give at least two.

1. `?- f(s(1), A).`
2. `?- f(s(s(1), two).`
3. `?- f(s(s(s(s(s(s(1)))))), C).`
4. `?- f(D, three).`

Problem XXIII

Write a Prolog predicate `sum_abs_diffs(List1, List2, Diffs)` which sums the absolute differences between two integer lists of the same length.
Example:

```
?- sum_abs_diffs([1,2,3], [5,4,2], X).
      X = 7 % abs(1-5) + abs(2-4) + abs(3-2)
```

Problem XXIV

Write a Prolog predicate `transpose(A, AT)` which transposes a rectangular matrix given in row-major order.
Example:

```
?- transpose([[1, 2], [3, 4]], AT).
        AT = [[1, 3], [2, 4]]
```

Problem XXV

Write Prolog predicates that given a database of countries and cities

```
% country(name, population (in thousands),
% capital).
country(sweden, 8823, stockholm).
country(usa, 221000, washington).
country(france, 56000, paris).
% city(name, in_country, population).
city(lund, sweden, 88).
city(paris, usa, 1).  % Paris, Texas.
```

Problem XXV...

Answer the following queries:

1. Which countries have cities with the same name as capitals of other countries?
2. In how many countries do more than $\frac{1}{3}$ of the population live in the capital?
3. Which capitals have a population more than 3 times larger than that of the secondmost populous city?

Problem XXV. . .

```
%country(name, population (in thousands), capital).
country(sweden, 8823, stockholm).
country(usa, 221000, washington).
country(france, 56000, paris).
country(denmark, 3400, copenhagen).
% city(name, in_country, population).
city(lund, sweden, 88).
city(new_york, usa, 5000).  % Paris, Texas.
city(paris, usa, 1).  % Paris, Texas.
city(copenhagen, denmark, 1200).
city(aarhus, denmark, 330).
city(odense, denmark, 120).
city(stockholm, sweden, 1300).
city(gothenburg, sweden, 350).
city(washington, usa, 3400).
city(paris, france, 2000).
```

Problem XXVI

Write a Prolog predicate that extracts all words immediately following "the" in a given list of words.
Example:

```
?- find([the, man, closed, the, door,
         of, the, house], X).
   X = [man, door, house]
```

Problem XXVII (Midterm Exam 372/04)

Write a Prolog predicate `dup` that duplicates each element of a list. Example:

```
?- dup([2,5,x], A).
   A = [2,2,5,5,x,x]
```

Problem XXVIII (Midterm Exam 372/04)

The following Prolog program evaluates constant expressions:

```
eval(A+B, V) :- eval(A, V1), eval(B, V2),
         V is V1 + V2.

eval(A*B, V) :- eval(A, V1), eval(B, V2),
         V is V1 * V2.

eval(X, X) :- integer(X).

?- eval(3*4+5, V).
   V = 17
```

Problem XXVIII. . . (Midterm Exam 372/04)

Modify the program so that it allows the expression to contain variables. Variable values should be taken from an environment (a list of variable/value pairs), like this:

```
?- eval([x=3,y=4], x*y+5, V).
   V = 17
?- eval([x=3], x*y+5, V).
   no
```

Problem XXIX (Midterm Exam 372/04)

Write a predicate `mult` which, for all pairs of numbers between 0 and 9, adds their product to the Prolog database. I.e., the following facts should be asserted:

```
times(0, 0, 0).   % 0∗0 = 0
times(0, 1, 0).   % 0∗1 = 0
...
times(9, 7, 63).   % 9∗7 = 63
times(9, 8, 72).   % 9∗8 = 72
times(9, 9, 81).   % 9∗9 = 81
```

The interaction should be as follows:

```
?- times(5,5,X).
no
?- mult.
yes
?- times(5,5,X).
X=25
```

Problem XXX (Midterm Exam 372/04)

Use a *2nd-order-predicate* to write a predicate `alltimes(L)` which, given the `times(X,Y,Z)` database above produces a list of all the multiplication facts:

```
?- alltimes(L).
L = [1*1=2,1*2=2,1*3=3,...,9*9=81].
```

Problem XXXI (Midterm Exam 372/04)

Show the results (`yes`/`no`) and resulting variable bindings for the following queries:

a) `?- f(g(X,X), h(Y,Y)) = f(g(Z), Z).`

b) `?- f(g(X,X), h(Y,Y)) = f(g(h(W,a),Z), Z).`

c) `?- f(g(X,X), h(_,_)) = f(g(h(W,a),Z), Z).`

d) `?- f(x(A,B),C ) = f( C,x(B,A)).`

Problem XXXII (Final Exam 372/04)

Given this Prolog predicate definition

```
mystery(L, B) :-
   member(X, L),
   append(A,[X],L),
   append(B,C,A),
   length(B,BL),
   length(C,CL),
   BL > CL.
```

what does the query

```
| ?- mystery([1,2,3,4,5],C), write(C), nl, fail.
```

print?

CSc 372

Comparative Programming Languages

32 : Prolog — Second-Order Predicates

Department of Computer Science
University of Arizona

collberg@gmail.com

Second-Order Programming

Second-Order Predicates

- When we ask a question in Prolog we will (if everything goes right) get an answer. **One** answer. We can if we want to ask Prolog to backtrack (using the semi-colon), but we will still only get one answer at a time.
- Furthermore, when we backtrack all the information gathered previously is lost.
- It isn't possible (in pure Prolog) to find the set of **all possible solutions** to a query.
- However, if we go outside pure Prolog (using the database manipulation features) we can construct procedures which collect all solutions to a query.
- They are called *second-order* because they deal with sets and the properties of sets, rather than about individual elements of sets.

Second-Order Predicates

- `setof(X,Goal,List)`
  - `List` is a collection of `X`s for which `Goal` is true.
  - `List` is sorted and contains no duplicates.
- `bagof(X,Goal,List)`
  - `List` is may contain duplicates.
- `setof` and `bagof` will fail if no `Goal`s succeed.
- `findall(X,Goal,List)`
  - `findall` will return `[]` if no `Goal`s succeed.

Examples

```
remove_duplicates(X, Y) :-
   setof(M, member(M,X), Y).

children(X,Kids) :-
   setof(C, father(X,C), Kids).
```

Uninstantiated Variables

- Consider `setof(X,Goal,List)` and `bagof(X,Goal,List)`.
- If there are uninstantiated variables in `Goal` which do not also appear in X, then a call to `setof` or `bagof` may backtrack, generating alternative values for `List`.
- If this is not the behavior you want, you can say

  ```
  Y ^ Goal
  ```

  meaning there exists a `Y` such that `Goal` is true, where `Y` is some Prolog term (usually, a variable).
- `findall` does this automatically.

Uninstantiated Variables…

- Consider this database:

```
foo(1,a).
foo(2,b).
foo(3,c).
```

- If we use both arguments of `foo` in our goal, we get what we expect:

```
| ?- findall(X/Y, foo(X,Y), L).
L = [1/a,2/b,3/c]
| ?- setof(X/Y, foo(X,Y), L).
L = [1/a,2/b,3/c]
| ?- bagof(X/Y, foo(X,Y), L).
L = [1/a,2/b,3/c]
```

Uninstantiated Variables. . .

- If we only use one of `foo`'s arguments in our goal, `findall` still gets us the expected result:

```
| ?- findall(X, foo(X,Y), L).
L = [1,2,3]
```

- But, `bagof` doesn't:

```
| ?- bagof(X, foo(X,Y), L).
L = [1]
Y = a ?  ;
L = [2]
Y = b ?  ;
L = [3]
Y = c
L = [1,2,3]
```

Uninstantiated Variables. . .

- So, instead we have to do:

```
| ?- bagof(X, Y^foo(X,Y), L).
L = [1,2,3]
```

SetOf — Drinkers

```
:- op(500, yfx, 'drinks').

john drinks whiskey.
martin drinks whiskey.
david drinks milk.
ben drinks milk.
helder drinks beer.
laurence drinks beer.
chris drinks coke.
louise drinks l_and_p.

?- setof(X, X drinks milk, S).
   X = _9109,
   S = [ben,david]
```

Implementing bagof

```
bagof(Item, Goal, _) :-
   assert(bag(marker)),
   Goal,
   assert(bag(Item)),
   fail.

bagof(_, _, Bag) :-
   retract(bag(Item)),
   collect(Item, [], Bag).

collect(marker, L, L).
collect(Item,ThisBag,FinalBag):-
   retract(bag(NextItem)),
   collect(NextItem,
      [Item|ThisBag], FinalBag).
```

Implementing `setof`

- `setof` is implemented as a call to `bagof` followed by a call to `sort` which puts the elements in order and removes duplicates.

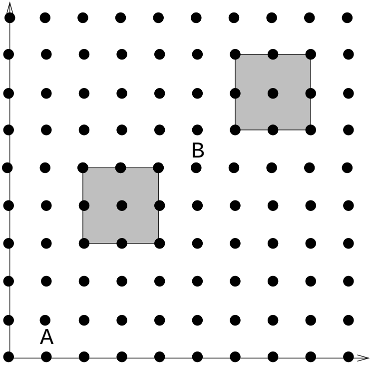
Lee's Algorithm

Lee's Algorithm

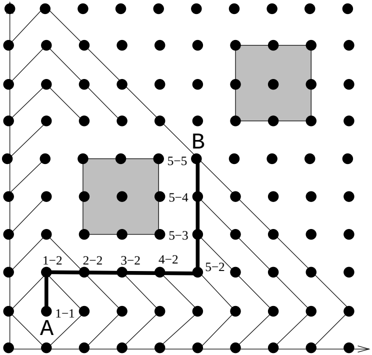
We are bext going to look a more involved example, an application from VLSI design. It uses the `setof` predicate to compute a shortest path between two points on a grid, subject to the conditions that

1. The path goes in the east-west-north-south direction only.
2. The path doesn't touch any obstacles.

- VLSI routing on a grid.
- Find a shortest Manhattan route between `A` and `B` that doesn't pass through any obstacles.



```
lee_route(A,B,Obstacles,Path) :-
   waves(B,[[A],[]],Obstacles,Waves),
   path(A,B,Waves,Path).
?- lee_route(1-1,5-5,[obst(2-3, 4-5),
      obst(6-6, 8-8)], P).
```



Lee's Algorithm. . .

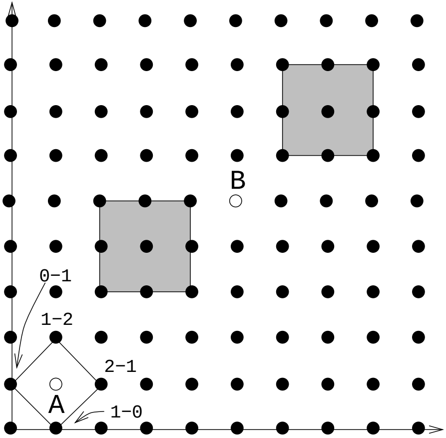
Lee's algorithm works in two stages:

1. First we generate a sequence of waves, where the first wave consists of the starting point itself.
2. Then we use the set of waves to find a shortest path.

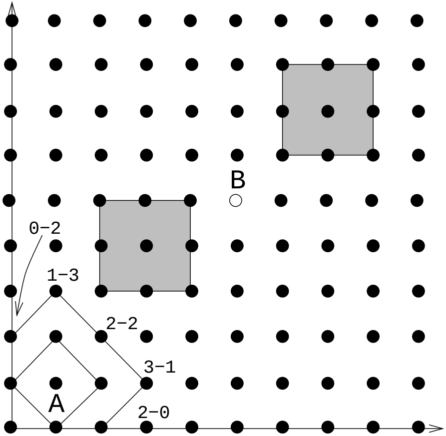
Lee's Algorithm. . .

- We start out with one wave which consists solely of the source point.
- From that point we generate all neighboring points. This forms the second wave.
- Each wave consists of points which are
  1. neighbors to points on the previous wave,
  2. not members of previous waves,
  3. not obstructed by any obstacles.
- We stop when the destination point is on the last generated wave.

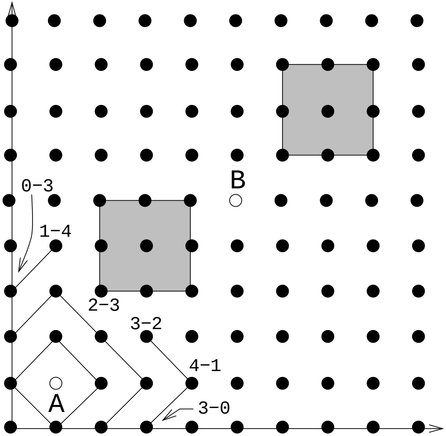
```
LastW  =  []
Wave   =  [1-1]
NextW  =  [0-1,1-0,1-2,2-1]
```



```
LastW   =   [1-1]
Wave    =   [0-1,1-0,1-2,2-1]
NextW   =   [0-0,0-2,1-3,2-0,2-2,3-1]
```



```
LastW  =  [0-1,1-0,1-2,2-1]
Wave   =  [0-0,0-2,1-3,2-0,2-2,3-1]
NextW  =  [0-3,1-4,3-0,3-2,4-1]
```



Lee's Algorithm...

```
waves(Destination,Wavessofar,Obstacles,Waves) :-
   Waves is a list of waves including
   Wavessofar (except, perhaps, it's last wave)
   that leads to Destination without crossing .
   Obstacles.

next_waves(Wave,LastWave,Obstacles,NextWave) :-
   Nextwave is the set of admissible points
   from Wave, that is excluding points from
   Lastwave, Wave, and points under Obstacles.
```

Lee's Algorithm. . .

- The first `wave`-rule (the recursive base case for `wave`) states that once the last generated wave contains the destination point, we're done generating waves.
- The second `wave`-rule simply generates the next wave (using `next_wave`), and then adds it to the beginning of the list of waves. Note that the list of waves is a *list-of-lists*.

Lee's Algorithm. . .

- `next_wave` takes three input parameters:
    1. `Wave` is the last generated wave.
    2. `LastWave` is the wave generated before the last wave.
    3. `Obstacels` is the list of obstacles.
- `next_wave` uses `setof` to generate the set of all *admissible* points. A point is admissible if it belongs to the next wave.

Lee's Algorithm...

```
waves(B,[Wave|Waves],Obstacles,Waves) :-
   member(B,Wave), !.
waves(B,[Wave,LastWave|LastWaves],
      Obstacles,Waves) :-
   next_wave(Wave,LastWave,Obstacles,NextWave),
   waves(B,[NextWave,Wave,LastWave|LastWaves],
      Obstacles,Waves).

next_wave(Wave,LastWave,Obstacles,NextWave) :-
   setof(X,admissible(X,Wave,LastWave,Obstacles),
      NextWave).
```

Lee's Algorithm. . .

`X` is **adjacent** to the points on `Wave` (i.e. `X` is a point on the next wave) if

- `X` is a neighbor to a point `X1` on the previous wave (`Wave`, that is).
- `X` is not obstructed by an obstacle.

Lee's Algorithm. . .

Notice that `adjacent` uses a **generate-and-test** scheme:

1. `member` & `neighbor` work together to generate new possible points:
    1. `member` generates points on the previous wave.
    2. `neighbor` uses the points generated by `member` to generate points which are neighbors to the points on the last wave.
2. `obstructed` weeds out generated point that are under an obstacle.

Lee's Algorithm. . .

`X` is an admissible point if

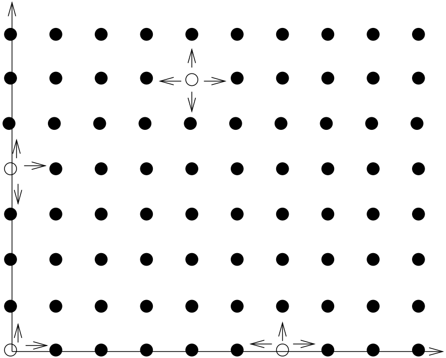
1. it is a neighbor of a point on the previous wave
2. it is not on any previous wave
3. is is not obstructed by an obstacle

```
admissible(X,Wave,LastWave,Obst) :-
   adjacent(X,Wave,Obst),
   not member(X,LastWave),
   not member(X,Wave).

adjacent(X,Wave,Obstacles) :-
   member(X1,Wave),
   neighbor(X1,X),
   not obstructed(X,Obstacles).
```

Lee's Algorithm. . .

- `next_to` takes a number `A` and returns `B=A+1` and `B=A-1`. `A-1` is returned only if the result is >0.
- `neighbor` uses `next_to` to generate neighboring points. The rules of `neighbor` state:
  1. The point `X2-Y` is a neighbor of point `X1-Y` if `X2 is X1+1`, or `X2=X1-1`. In other words, the first `neighbor` rule generates the points immediately above and below a given point.
  2. The point `X-Y2` is a neighbor of point `X-Y1` if `Y2 is Y1+1`, or `Y2=Y1-1`. In other words, the second `neighbor` rule generates the points immediately to the left and right of a given point.

```
neighbor(X1-Y,X2-Y):- next_to(X1,X2).
neighbor(X-Y1,X-Y2):- next_to(Y1,Y2).

next_to(A,B) :- B is A+1.
next_to(A,B) :- A > 0, B is A-1.
```



Lee's Algorithm…

- `obstructed(Point,Obstacles)` checks to see if the point is on the perimeter of any of the obstacles in the list of obstacles `Obstacles`.
- The rule `obstructs(Point, Obstacle)` checks to see if the point is on the `perimeter` of the obstacle.

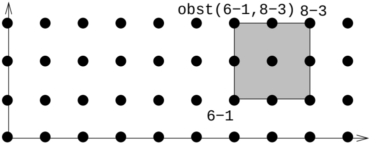
Note that `obstructed` is another generate-and-test procedure. `member` generates one obstacle at a time from this list, and `obstructs` checks to see if that obstacle obstructs the point.

Lee's Algorithm. . .

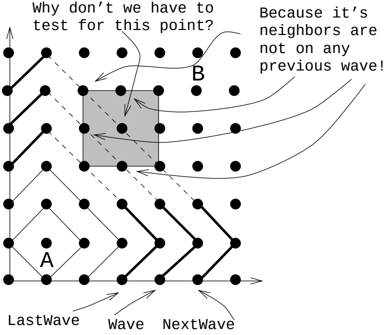
- `obstructed(Point,Obstacles)` checks to see if the point is on the perimeter of any of the obstacles in the list of obstacles `Obstacles`.
- The rule `obstructs(Point, Obstacle)` checks to see if the point is on the `perimeter` of the obstacle.

Note that `obstructed` is another generate-and-test procedure. `member` generates one obstacle at a time from this list, and `obstructs` checks to see if that obstacle obstructs the point.

```
% Generate an obstacle, then test
% if it obstructs a point Pt.
obstructed(Pt,Obsts) :-
   member(Obst,Obsts), obstructs(Pt,Obst).
obstructs(X-Y,obst(X-Y1,X2-Y2)) :-
   Y1=<Y, Y=<Y2.  % X-Y on bottom edge.
obstructs(X-Y,obst(X1-Y1,X-Y2)) :- Y1=<Y,Y=<Y2.
obstructs(X-Y,obst(X1-Y,X2-Y2)) :- X1=<X,X=<X2.
obstructs(X-Y,obst(X1-Y1,X2-Y)) :- X1=<X,X=<X2.
```



- Why do we only need to check the perimeter? Shouldn't we have to check if a point lies *inside* an object as well?
- No, such points will never be considered. Their neighbors (which are on a perimeter) cannot be on a previous wave:



Lee's Algorithm. . .

The last part of the algorithm is to construct the actual path from the list of waves. The procedure `path` does this for us.

1. `path` starts by looking in the last wave for a neighbor of the destination node. In our example, the destination node is `5-5`, and a neighbor of `5-5` in the last wave is the node `5-4`.
2. `path` next looks for a neighbor for the new node in the next wave. Our example yields node `5-3` which is a neighbor of node `5-4`.
3. Eventually we'll get to the last wave which only contains the source node, in our case node `1-1`.

Lee's Algorithm. . .

```
Waves = [[0-7,1-8,2-7,3-6,5-4,6-3,7-0,7-2,8-1],
         [0-6,1-7,2-6,5-3,6-0,6-2,7-1],
         [0-5,1-6,5-0,5-2,6-1],
         [0-4,1-5,4-0,4-2,5-1],
         [0-3,1-4,3-0,3-2,4-1],
         [0-0,0-2,1-3,2-0,2-2,3-1],
         [0-1,1-0,1-2,2-1],
         [1-1]]

path(A,A,Waves,[A]) :- !.
path(A,B,[Wave|Waves],[B|Path]) :-
   member(B1,Wave),
   neighbor(B,B1), !,
   path(A,B1,Waves,Path).
```

Readings and References

- Read `Clocksin & Mellish, pp.  156--158`.

homework

Exercise

Write Prolog predicates that given a database of countries and cities

```
% country(name, population, capital).
country(sweden, 8823, stockholm).
country(usa, 221000, washington).
country(france, 56000, paris).
% city(name, in_country, population).
city(lund, sweden, 88).
city(paris, usa, 1).  % Paris, Texas.
```

Exercise...

answer the following queries:

1. Which countries have cities with the same name as capitals of other countries?
2. In how many countries do more than $\frac{1}{3}$ of the population live in the capital?
3. Which capitals have a population more than 3 times larger than that of the secondmost populous city?

CSc 372

Comparative Programming Languages

33 : Prolog — Grammars

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

Prolog Grammar Rules

Prolog Grammar Rules

- A DCG (definite clause grammar) is a phrase structure grammar annotated by Prolog variables.
- DCGs are translated by the Prolog interpreter into normal Prolog clauses.
- Prolog DCG:s can be used for generation as well as parsing. I.e. we can run the program backwards to generate sentences from the grammar.

Prolog Grammar Rules. . .

```
s      --> np, vp.
vp     --> v, np.
vp     --> v.
np     --> n.
n      --> [john].    n      --> [lisa].
n      --> [house].
v      --> [died].    v      --> [kissed].
```

```
?- s([john, kissed, lisa], []).
   yes
?- s([lisa, died], []).
   yes
?- s([kissed, john, lisa], []).
   no
```

Prolog Grammar Rules. . .

```
?- s(A, []).
   A = [john,died,john] ;
   A = [john,died,lisa] ;
   A = [john,died,house] ;
   A = [john,kissed,john] ;
   A = [john,kissed,lisa] ;
   A = [john,kissed,house] ;
   A = [john,died] ;
   A = [john,kissed] ;
   A = [lisa,died,john] ;
   A = [lisa,died,lisa] ;
   A = [lisa,died,house] ;
   A = [lisa,kissed,house] ;
   A = [lisa,died] ;
```

Implementing Prolog Grammar Rules

- Prolog turns each grammar rule into a clause with one argument.
- The rule $S \rightarrow NP\ VP$ becomes

```
s(Z) :- np(X), vp(Y), append(X,Y,Z).
```

- This states that `Z` is a sentence if `X` is a noun phrase, `Y` is a verb phrase, and `Z` is `X` followed by `Y`.

Implementing Prolog Grammar Rules. . .

```
s(Z) :- np(X), vp(Y), append(X,Y,Z).
np(Z) :- n(Z).
vp(Z) :- v(X), np(Y), append(X,Y,Z).
vp(Z) :- v(Z).
n([john]).  n([lisa]).  n([house]).
v([died]).  v([kissed]).
```

```
?- s([john,kissed,lisa]).
   yes
?- s(S).
   S = [john,died,john] ;
   S = [john,died,lisa] ; ...
```

Implementing Prolog Grammar Rules. . .

- The `append`'s are expensive — Prolog uses difference lists instead.
- The rule

  ```
  s(A,B) :- np(A,C), vp(C,B).
  ```

  says that there is a sentence at the beginning of A (with B left over) if there is a noun phrase at the beginning of A (with C left over), and there is a verb phrase at the beginning of C (with B left over).

Implementing Prolog Grammar Rules. . .

```
s(A,B)    :- np(A,C), vp(C,B).
np(A,B)   :- n(A,B).
vp(A,B)   :- v(A,C), np(C,B).
vp(A,B)   :- v(A,B).
n([john|R],R). n([lisa|R],R).
v([died|R],R). v([kissed|R],R).
```

```
?- s([john,kissed,lisa], []).
   yes

?- s([john,kissed|R], []).
   R = [john] ;
   R = [lisa] ;...
```

Generating Parse Trees

- DCGs can build parse trees which can be used to construct a semantic interpretation of the sentence.
- The tree is built bottom-up, when Prolog returns from recursive calls. We give each phrase structure rule an extra argument which represents the node to be constructed.

Generating Parse Trees...

```
s(s(NP,VP))      --> np(NP), vp(VP).
vp(vp(V, NP))    --> v(V), np(NP).
vp(vp(V))        --> v(V).
np(np(N))        --> n(N).
n(n(john))       --> [john].
n(n(lisa))       --> [lisa].
n(n(house))      --> [house].
v(n(died))       --> [died].
v(n(kissed))     --> [kissed].
```

Generating Parse Trees...

- The rule

  ```
  s(s(NP,VP)) --> np(NP), vp(VP).
  ```

  says that the top-level node of the parse tree is an s with the sub-trees generated by the np and vp rules.

```
?- s(S, [john, kissed, lisa], []).
S=s(np(n(john)),vp(n(kissed),np(n(lisa))))
?- s(S, [lisa, died], []).
S=s(np(n(lisa)),vp(n(died)))
?- s(S, [john, died, lisa], []).
S=s(np(n(john)),vp(n(died),np(n(lisa))))
```

Generating Parse Trees…

- We can of course run the rules backwards, turning parse trees into sentences:

```
?- s(s(np(n(john)),vp(n(kissed),
        np(n(lisa)))), S, []).
      S=[john, kissed, lisa]
```

Ambiguity

Ambiguity

- An ambigous sentence is one which can have more than one meaning.

Lexical ambiguity:

homographic

- spelled the same
- *bat* (wooden stick/animal)
- *import* (noun/verb)

polysemous

- different but related meanings
- *neck* (part of body/part of bottle/narrow strip of land)

homophonic

- sound the same
- to/too/two

Ambiguity. . .

Syntactic ambiguity:

- More than one parse (tree).
- Many missiles have many war-heads.

- "Duck" can be either a verb or a noun.
- "her" can either be a determiner (as in "her book"), or a noun: "I liked her dancing".

Ambiguity. . .

```
s(s(NP,VP)) --> np(NP), vp(VP).
vp(vp(V, NP)) --> v(V), np(NP).
vp(vp(V, S)) --> v(V), s(S).
vp(vp(V)) --> v(V).
np(np(Det,N)) --> det(Det), n(N).
np(np(N)) --> n(N).
n(n(i)) --> [i].
n(n(duck)) --> [duck].
v(v(duck)) --> [duck].
v(v(saw)) --> [saw].  n(n(saw)) --> [saw].
n(n(her)) --> [her].
det(det(her)) --> [her].
```

```
?- s(S, [i, saw, her, duck], []).
```

DCG Applications

Pascal Declarations

```
?- decl([const, a, =, 5, ;,
        var, x, :, 'INTEGER', ;], []).
   yes
?- decl([const, a, =, a, ;, var, x,
        :, 'INTEGER', ;], []).
   no

decl --> const_decl, type_decl,
         var_decl, proc_decl.
```

Pascal Declarations

```
% Constant declarations
const_decl --> [ ].
const_decl -->
   [const], const_def, [;], const_defs.

const_defs --> [ ].
const_defs --> const_def, [;], const_defs.
const_def --> identifier, [=], constant.

identifier --> [X], {atom(X)}.
constant --> [X], {(integer(X); float(X))}.
```

Pascal Declarations. . .

```
% Type declarations
type_decl --> [ ].
type_decl --> [type], type_def, [;], type_defs.

type_defs --> [ ].
type_defs --> type_def, [;], type_defs.
type_def --> identifier, [=], type.

type --> ['INTEGER']. type --> ['REAL'].
type --> ['BOOLEAN']. type --> ['CHAR'].
```

Pascal Declarations. . .

```
% Variable decleclarations
var_decl --> [ ].
var_decl --> [var], var_def, [;], var_defs.

var_defs --> [ ].
var_defs --> var_def, [;], var_defs.
var_def --> id_list, [:], type.

id_list --> identifier.
id_list --> identifier, [’,’], id_list.
```

Pascal Declarations. . .

```
% Procedure declarations
proc_decl --> [ ].
proc_decl --> proc_heading, [;], block.
proc_heading --> [procedure], identifier,
            formal_param_part.
formal_param_part --> [ ].
formal_param_part --> ['('],
            formal_param_section, [')'].
formal_param_section --> formal_params.
formal_param_section --> formal_params, [;],
   formal_param_section.
formal_params --> value_params.
formal_params --> variable_params.
value_params --> var_def.
variable_params --> [var], var_def.
```

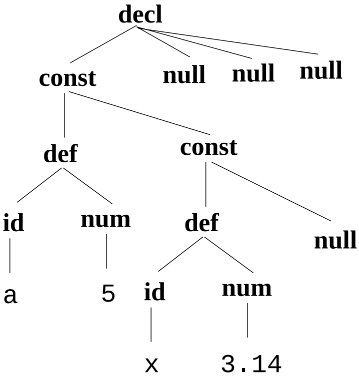
Pascal Declarations – Building Trees

```
decl(decl(C, T, V, P)) -->
    const_decl(C), type_decl(T),
    var_decl(V), proc_declaration(P).

const_decl(const(null)) --> [ ].
const_decl(const(D, Ds)) -->
    [const], const_def(D), [;], const_defs(Ds).
```

Pascal Declarations – Building Trees…

```
const_defs(null) --> [ ].
const_defs(const(D, Ds)) -->
   const_def(D), [;], const_defs(Ds).

const_def(def(I, C)) --> ident(I), [=], const(C).

ident(id(X)) --> [X], {atom(X)}.
const(num(X)) --> [X], {(integer(X); float(X))}.
```

Pascal Declarations – Example Parse



Pascal Declarations – Example Parse. . .

```
?- decl(S, [const, a, =, 5, ;, x, =, 3.14, ;], []).

   S = decl(
          const(def(id(a),num(5)),
             const(def(id(x),num(3.14)),
                   null)),
          null,null,null)
```

Number Conversion

```
?- number(V, [sixty, three], []).
   V = 63
?- number(V,[one,hundred,and,fourteen],[]).
   V = 114
?- number(V,[nine,hundred,and,ninety,nine],[]).
   V = 999
?- number(V, [fifty, ten], []).
   no
```

Number Conversion…

```
number(0) --> [zero].
number(N) --> xxx(N).

xxx(N) --> digit(D), [hundred], rest_xxx(N1),
            {N is D * 100+N1}.
xxx(N) --> xx(N).

rest_xxx(0) --> [ ].  rest_xxx(N) --> [and], xx(N).

xx(N) --> digit(N).
xx(N) --> teen(N).
xx(N) --> tens(T), rest_xx(N1), {N is T+N1}.

rest_xx(0) --> [ ].  rest_xx(N) --> digit(N).
```

Number Conversion…

```
digit(1) --> [one].      teen(10) --> [ten].
digit(2) --> [two].      teen(11) --> [eleven].
digit(3) --> [three].    teen(12) --> [twelve].
digit(4) --> [four].     teen(13) --> [thirteen].
digit(5) --> [five].     teen(14) --> [fourteen].
digit(6) --> [six].      teen(15) --> [fifteen].
digit(7) --> [seven].    teen(16) --> [sixteen].
digit(8) --> [eight].    teen(17) --> [seventeen].
digit(9) --> [nine].     teen(18) --> [eighteen].
                         teen(19) --> [nineteen].
tens(20) --> [twenty].   tens(30) --> [thirty].
tens(40) --> [forty].    tens(50) --> [fifty].
tens(60) --> [sixty].    tens(70) --> [seventy].
tens(80) --> [eighty] .  tens(90) --> [ninety].
```

Expression Evaluation

- Evaluate infix arithmetic expressions, given as character strings.

```
?- expr(X, "234+345*456", []).
   X = 157554

expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(Z) --> term(Z).

term(Z) --> num(X), "*", term(Y), {Z is X * Y}.
term(Z) --> num(X), "/", term(Y), {Z is X /Y }.
term(Z) --> num(Z).
```

Expression Evaluation. . .

- Prolog grammar rules are equivalent to recursive descent parsing. Beware of left recursion!
- Anything within curly brackets is “normal” Prolog code.

```
num(C) --> "+", num(C).
num(C) --> "-", num(X), {C is -X}.
num(X) --> int(0, X).

int(L, V) --> digit(C), {V is L * 10 +C}.
int(L, X) --> digit(C), {V is L* 10 +C},
          int(V, X).

digit(X) --> [C], {"0" =< C, C =< "9",X is C-"0"}.
```

Machine Translation

English to Maaori Translation

```
e2m(E, M) :-
   english_s(PL, E, []),
   maori_s(PL, M, []).

| ?- e2m([a, man, likes, beer], M).
M = [ka,pai,a,waipirau,ki,teetahi,tangata]
| ?- e2m([every, man, likes, beer], M).
M = [ka,pai,a,waipirau,ki,kotoa,tangata]
| ?- e2m([every, man, likes, beer], M).
M = [ka,pai,a,waipirau,ki,kotoa,tangata]
| ?- e2m(E, [ka,pai,te,waipirau,ki,teetahi,tangata]).
E = [a,man,likes,beer]
```

English to Predicate Logic

```
:- op(500, xfy, &).
:- op(500, xfy, =>).

english_s(Meaning) -->
    english_np(Who, Assn, Meaning),
    english_vp(Who, Assn).

english_det(Who, Prop, Assn,
            exists(Who, Prop & Assn)) --> [a].
english_det(Who, Prop, Assn,
            all(Who, Prop => Assn)) --> [every].

english_np(Who, Assn, Assn) -->
   english_noun(Who, Who).
```

```
english_np(Who, Assn, Meaning) -->
   english_det(Who, Prop, Assn, Meaning),
   english_noun(Who, Prop).

english_noun(Who, man(Who)) --> [man].
english_noun(beer, beer) --> [beer].
english_noun(john, john) --> [john].

english_vp(Who, Meaning) -->
   english_intrans_v(Who, Meaning).
english_vp(Who, Meaning) -->
   english_trans_v(Who, What, Meaning),
   english_np(What, Assn, Assn).

english_intrans_v(Who, sleeps(Who)) --> [sleeps].

english_trans_v(Who, What,
                likes(Who, What)) --> [likes].
```

Maaori to Predicate Logic

```
maori_s(Meaning) -->
    maori_trans_vp(Who, Assn),
    maori_pp(Who, Assn, Meaning).

maori_det   --> [a].     % pers
maori_det   --> [te].    % the
maori_det   --> [ngaa].  % the-pl

maori_quant(Who, Prop, Assn,
            exists(Who, Prop & Assn)) --> [teetahi].
maori_quant(Who, Prop, Assn,
            all(Who, Prop => Assn))   --> [kotoa].

maori_np(Who, Meaning, Meaning) -->
   maori_det,
   maori_noun(Who, Who).
```

```
maori_np(Who, Assn, Meaning) -->
   maori_quant(Who, Prop, Assn, Meaning),
   maori_noun(Who, Prop).

maori_np(Who, Assn, Meaning) -->
   maori_det,
   maori_noun(Who, Prop),
   maori_quant(Who, Prop, Assn, Meaning).

maori_pp(Who, Assn, Meaning) -->
   [ki],
   maori_np(Who, Assn, Meaning).

maori_noun(Who, man(Who)) --> [tangata].  % man
maori_noun(Who, man(Who)) --> [tangaata]. % men
maori_noun(beer, beer)   --> [waipirau].
maori_noun(john, john)   --> [hone].
```

```
maori_intrans_v(Who, sleeps(Who)) --> [sleeps].

maori_trans_vp(Who, Assn) -->
   maori_tense,
   maori_trans_v(Who, What, Assn),
   maori_np(What, Assn, Assn).

maori_tense --> [ka].
maori_trans_v(Who, What, likes(Who, What)) --> [pai].
```

Summary

Summary

- Read Clocksin & Mellish, Chapter 9.
- Grammar rule syntax:
  - A grammar rule is written `LHS --> RHS`. The left-hand side (`LSH`) must be a non-terminal symbol, the right-hand side (`RHS`) can be a combination of terminals, non-terminals, and Prolog goals.
  - Terminal symbols (words) are in square brackets: `n --> [house]`.
  - More than one terminal can be matched by one rule: `np --> [the,house]`.

Summary. . .

- Grammar rule syntax (cont):
  - Non-terminals (syntactic categories) can be given extra arguments: `s(s(N,V)) --> np(N),vp(V).`.
  - Normal Prolog goals can be embedded within grammar rules: `int(C) --> [C],{integer(C)}`.
  - Terminals, non-terminals, and Prolog goals can be mixed in the right-hand side: `x --> [y], z, {w}, [r], p.`
- Beware of left recursion! `expr --> expr ['+'] expr` will recurse infinitely. Rules like this will have to be rewritten to use right recursion.

Exercise

Exercise

- Write a program which uses Prolog Grammar Rules to convert between English time expressions and a 24-hour clock ("Military Time").
- You may assume that the following definitions are available:

```
digit(1) --> [one].  ....
digit(9) --> [nine].
teen(10) --> [ten].  ....
teen(19) --> [nineteen].
tens(20) --> [twenty].  ....
tens(90) --> [ninety].
```

```
?- time(T, [eight, am], []).
   T = 8:0 % Or, better, 8:00
```

Exercise. . .

```
?- time(T, [eight, thirty, am], []).
   T = 8:30
?- time(T,[eight,fifteen,am],[]).
   T = 8:15
?- time(T,[eight,five,am],[]).
   no
?- time(T,[eight,oh,five,am],[]).
   T = 8:5 % Or, better, 8:05
?- time(T,[eight,oh,eleven,am],[]).
   no
?- time(T,[eleven,thirty,am],[]).
   T = 11:30
?- time(T,[twelve,thirty,am],[]).
   T = 0:30 % !!!
```

Exercise. . .

```
?- time(T,[eleven,thirty,pm],[]).
   T = 23:30
?- time(T,[twelve,thirty,pm],[]).
   T = 12:30 % !!!
?- time(T,[ten,minutes,to,four,am],[]).
   T = 3:50
?- time(T,[ten,minutes,past,four,am],[]).
   T = 4:10
?- time(T,[quarter,to,four,pm],[]).
   T = 15:45
?- time(T,[quarter,past,four,pm],[]).
   T = 16:15
?- time(T,[half,past,four,pm],[]).
   T = 16:30
```