Prolog:
Programming in Logic

with some mention of Datalog and Constraint Logic Programming

The original declarative programming language

- Courses in programming languages …
    - Prolog is always the declarative language they teach.
    - (imperative, functional, object-oriented, declarative)

- Alain Colmeraeur & Philippe Roussel, 1971-1973
    - With help from theorem proving folks such as Robert Kowalski
    - Original project: Type in French statements & questions
        - Computer needed NLP and deductive reasoning
    - Efficiency by David Warren, 1977 (compiler, virtual machine)
    - Colmerauer & Roussel wrote 20 years later:
      "Prolog is so simple that one has the sense that sooner or later someone had to discover it … that period of our lives remains one of the happiest in our memories.
    - "We have had the pleasure of recalling it for this paper over almonds accompanied by a dry martini."

Prolog vs. ECLiPSe

- Most common free Prolog implementation is SWI Prolog.
  - Very nice, though faster ones are for sale (e.g., SICSTUS Prolog).
- To run Prolog, you can just run ECLiPSe!
  - ECLiPSe is a perfectly good Prolog implementation, although so far we've concentrated only on its "extra" features.

Prolog vs. ECLiPSe

Constraint programming	Logic programming (e.g., Prolog)	Constraint logic programming (e.g., ECLiPSe)
Efficient: Variable ordering Value ordering Constraint joining and propagation **But:** Encoding is annoying Variables limited to finite sets, ints, reals	**Expressive:** Subroutines Recursion Variable domains are “terms” (including lists and trees) **But:** Simple, standard solver: backtracking and unification	**Combo:** Tries to combine best of both worlds Later on we’ll see how

Prolog as constraint programming

(Person, Food)

		🟩	🟩
	🟩		🟩
🟩		🟩	

Person	Food
sam	dal
sam	curry
josie	samosas
josie	curry
rajiv	burgers
rajiv	dal

- The above shows an ordinary constraint between two variables: Person and Food
- Prolog makes you name this constraint.
  Here's a program that defines it:
  - eats(sam, dal). eats(josie, samosas).
  - eats(sam, curry). eats(josie, curry).
  - eats(rajiv, burgers). eats(rajiv, dal). …
- Now it acts like a subroutine! At the Prolog prompt you can type
  - eats(Person1, Food1). % constraint over two variables
  - eats(Person2, Food2). % constraint over two **other** variables

Simple constraints in Prolog

- Here’s a program defining the “eats” constraint:
  - eats(sam, dal).   eats(josie, samosas).
  - eats(sam, curry).   eats(josie, curry).
  - eats(rajiv, burgers).   eats(rajiv, dal). …
  - Now at the Prolog prompt you can type
    - eats(Person1, Food1).   % constraint over two variables
    - eats(Person2, Food2).   % constraint over two **other** variables

- To say that Person1 and Person2 must eat a common food, conjoin two constraints with a comma:
  - eats(Person1, Food), eats(Person2, Food).
  - Prolog gives you possible solutions:
    - Person1=sam, Person2=josie, Food=curry
    - Person1=josie, Person2=sam, Food=curry   …

Actually, it will start with solutions where Person1=sam, Person2=sam. How to fix?

Color coding in these slides

- eats(sam, dal). eats(josie, samosas).
- eats(sam, curry). eats(josie, curry).
- eats(rajiv, burgers). eats(rajiv, dal). …

Your program file (compiled)
Sometimes called the "database"

"Query" that you type interactively

- eats(Person1, Food), eats(Person2, Food).

Prolog's answer

- Person1=sam, Person2=josie, Food=curry
- Person1=josie, Person2=sam, Food=curry …

Simple constraints in Prolog

- Here's a program defining the "eats" constraint:
  - eats(sam, dal). eats(josie, samosas).
  - eats(sam, curry). eats(josie, curry).
  - eats(rajiv, burgers). eats(rajiv, dal). …
  - Now at the Prolog prompt you can type
    - eats(Person1, Food1). % constraint over two variables
    - eats(Person2, Food2). % constraint over two **other** variables

- To say that Person1 and Person2 must eat a common food, conjoin two constraints with a comma:
  - eats(Person1, Food), eats(Person2, Food).
  - Prolog gives you possible solutions:
    - Person1=sam, Person2=josie, Food=curry
    - Person1=josie, Person2=sam, Food=curry …

Actually, it will start with solutions where Person1=sam, Person2=sam. How to fix?

Queries in Prolog

These things you type at the prompt are called "queries."

- Prolog answers a query as "Yes" or "No" according to whether it can find a satisfying assignment.
- If it finds an assignment, it prints the first one before printing "Yes."
- You can press Enter to accept it, in which case you're done, or ";" to reject it, causing Prolog to backtrack and look for another.

  - eats(Person1, Food1). % constraint over two variables
  - eats(Person2, Food2). % constraint over two **other** variables

- eats(Person1, Food), eats(Person2, Food).
- Prolog gives you possible solutions:
  - Person1=sam, Person2=josie, Food=curry [press ";"]
  - Person1=josie, Person2=sam, Food=curry …

Constants vs. Variables

- Here's a program defining the "eats" constraint:
  - eats(sam, dal). eats(josie, samosas).
  - eats(sam, curry). eats(josie, curry).
  - eats(rajiv, burgers). …
  - Now at the Prolog prompt you can type
    - eats(Person1, Food1). % constraint over two variables
    - eats(Person2, Food2). % constraint over two **other** variables

- Nothing stops you from putting constants into constraints:
    - eats(josie, Food). % what Food does Josie eat? (2 answers)
    - eats(Person, curry). % what Person eats curry? (2 answers)
    - eats(josie, Food), eats(Person, Food). % who'll share what with Josie?
      - Food=curry, Person=sam

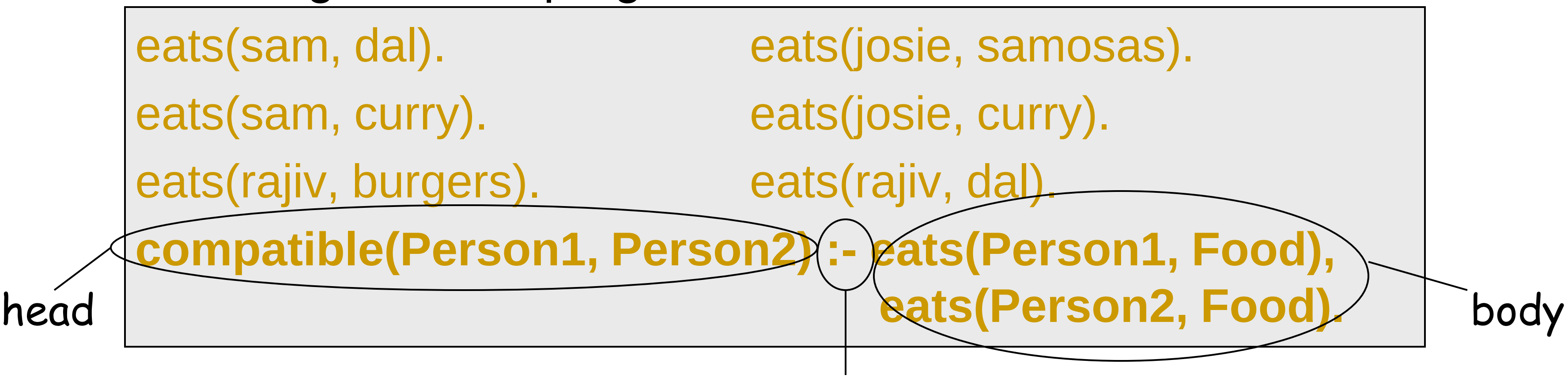
Constants vs. Variables

- Variables start with A,B,…Z or underscore:
  - Food, Person, Person2, _G123
- Constant "atoms" start with a,b,…z or appear in single quotes:
  - josie, curry, 'CS325'
  - Other kinds of constants besides atoms:
    - Integers -7, real numbers 3.14159, the empty list []
    - eats(josie,curry) is technically a constant **structure**

- Nothing stops you from putting constants into constraints:
    - eats(josie, Food). % what Food does Josie eat? (2 answers)
    - eats(Person, curry). % what Person eats curry? (2 answers)
    - eats(josie, Food), eats(Person, Food). % who'll share what with Josie?
      - Food=curry, Person=sam

Rules in Prolog

- Let’s augment our program with a new constraint:



```
eats(sam, dal).                        eats(josie, samosas).
eats(sam, curry).                      eats(josie, curry).
eats(rajiv, burgers).                  eats(rajiv, dal).
compatible(Person1, Person2) :- eats(Person1, Food),
                                eats(Person2, Food).
```

means “if” – it’s supposed to look like “←”

  - “Person1 and Person2 are compatible if there exists some Food that they both eat.”
  - “One way to satisfy the head of this rule is to satisfy the body.”
  - You type the query: compatible(rajiv, X). Prolog answers: X=sam.
    - Prolog doesn’t report that Person1=rajiv, Person2=sam, Food=dal. These act like local variables in the rule. It already forgot about them.

Rules in Prolog

- Let’s augment our program with a new constraint:

```
eats(sam, dal).                eats(josie, samosas).
eats(sam, curry).              eats(josie, curry).
eats(rajiv, burgers).          eats(rajiv, dal).
compatible(Person1, Person2) :- eats(Person1, Food),
                                eats(Person2, Food).
compatible(Person1, Person2) :- watches(Person1, Movie),
                                watches(Person2, Movie).
compatible(hal, Person2) :- female(Person2), rich(Person2).
```

  - “One way to satisfy the head of this rule is to satisfy the body.”

why only “one way”? Why not “if and only if”?

allusion to movie *Shallow Hal*;
shows that constants can appear in rules

The Prolog solver

- Prolog's solver is incredibly simple.
- eats(sam,X).
  - Iterates <u>in order</u> through the program's "eats" clauses.
  - First one to match is eats(sam,dal).
    so it returns with X=dal.
  - If you hit semicolon, it backtracks and continues:
    Next match is eats(sam,curry).
    so it returns with X=curry.

The Prolog solver

- Prolog’s solver is incredibly simple.
- eats(sam,X).
- eats(sam,X), eats(josie,X).
  - It satisfies 1st constraint with X=dal. Now X is assigned.
  - Now to satisfy 2nd constraint, it must prove eats(josie,dal). No!
  - So it backs up to 1st constraint & tries X=curry (sam’s other food).
  - Now it has to prove eats(josie,curry). Yes!
  - So it is able to return X=curry. What if you now hit semicolon?
- eats(sam,X), eats(Companion, X).
  - What happens here?
  - What variable ordering is being used? Where did it come from?
  - What value ordering is being used? Where did it come from?

The Prolog solver

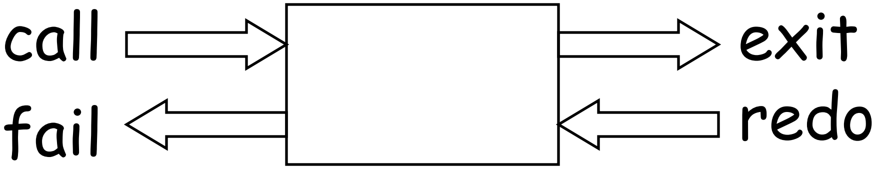
- Prolog's solver is incredibly simple.
- eats(sam,X).
- eats(sam,X), eats(josie,X).
- eats(sam,X), eats(Companion, X).
- compatible(sam,Companion).
  - This time, first clause that matches is
    compatible(Person1, Person2) :- eats(Person1, Food),
    eats(Person2, Food).
  - "Head" of clause matches with Person1=sam, Person2=Companion.
  - So now we need to satisfy "body" of clause:
    eats(sam,Food), eats(Companion,Food).
    Look familiar?
  - We get Companion=rajiv.

The Prolog solver

- Prolog’s solver is incredibly simple.
- eats(sam,X).
- eats(sam,X), eats(josie,X).
- eats(sam,X), eats(Companion, X).
- compatible(sam,Companion).
- compatible(sam,Companion), female(Companion).
    - **compatible(Person1, Person2) :- eats(Person1, Food),**
      eats(Person2, Food).
    - Our first try at satisfying 1st constraint is Companion=rajiv (as before).
        - But then 2nd constraint is female(rajiv). which is presumably false.
    - So we backtrack and look for a different satisfying assignment of the first constraint: Companion=josie.
        - Now 2nd constraint is female(josie). which is presumably true.
        - We backtracked into this **compatible** clause (food) & retried it.
        - No need yet to move on to the next **compatible** clause (movies).

Backtracking and Beads

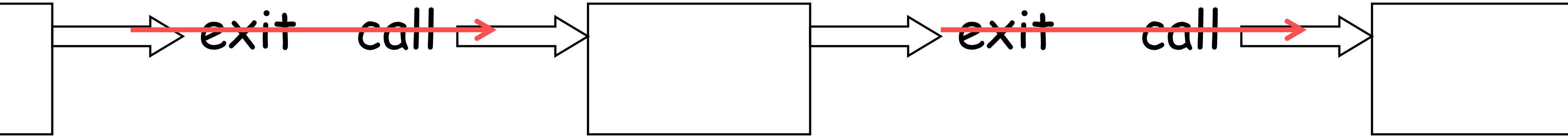
- Each Prolog constraint is like a “bead” in a string of beads:



- Each constraint has four ports: call, exit, redo, fail

Backtracking and Beads

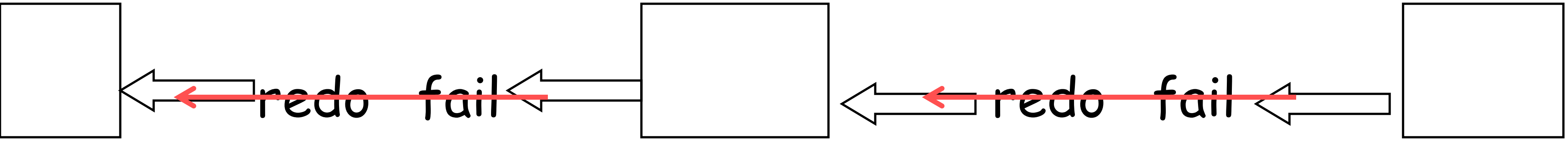
- Each Prolog constraint is like a “bead” in a string of beads:



  - Each constraint has four ports: *call*, *exit*, *redo*, *fail*
  - *exit* ports feed forward into *call* ports

Backtracking and Beads

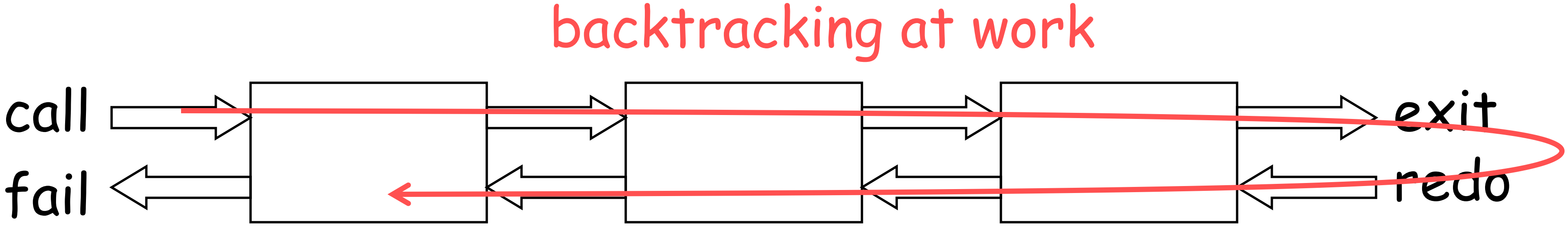
- Each Prolog constraint is like a “bead” in a string of beads:



- Each constraint has four ports: *call*, *exit*, *redo*, *fail*
- *exit* ports feed forward into *call* ports
- *fail* ports feed back into *redo* ports

Backtracking and Beads

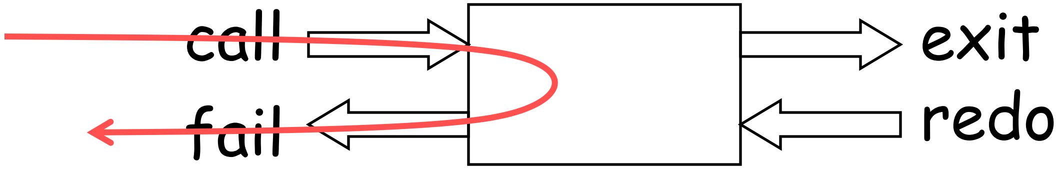
- Each Prolog constraint is like a “bead” in a string of beads:



  - Each constraint has four ports: *call*, *exit*, *redo*, *fail*
  - *exit* ports feed forward into *call* ports
  - *fail* ports feed back into *redo* ports

Backtracking and Beads

- Each Prolog constraint is like a “bead” in a string of beads:

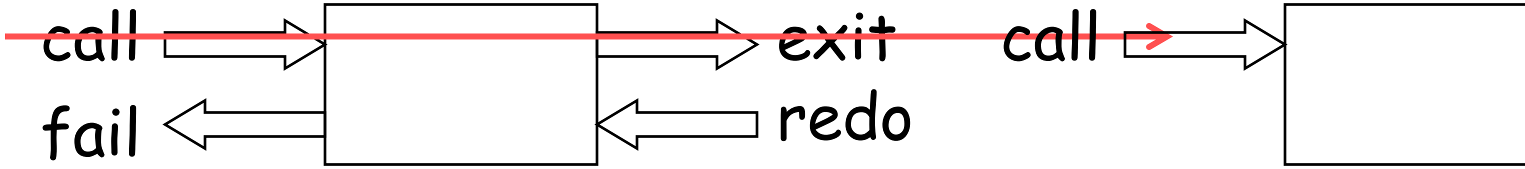


no way to satisfy this constraint given the assignments so far – so first call fails

How disappointing. Let’s try a happier outcome.

Backtracking and Beads

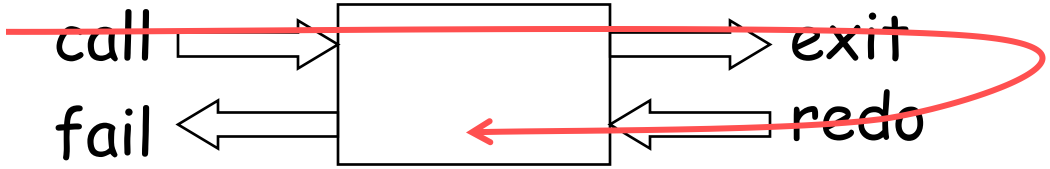
- Each Prolog constraint is like a “bead” in a string of beads:



we satisfy this constraint, making additional assignments, and move on …

Backtracking and Beads

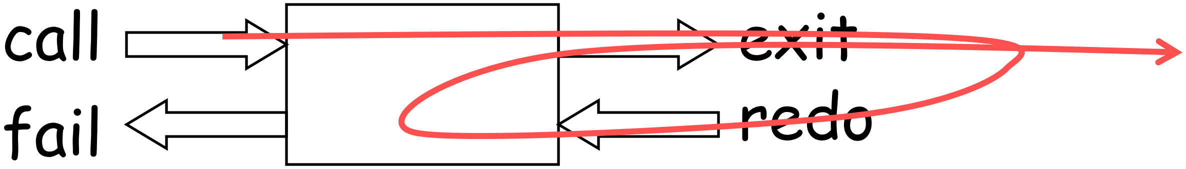
- Each Prolog constraint is like a "bead" in a string of beads:



we satisfy this constraint, making additional assignments, and move on …
but if our assignments cause later constraints to fail, Prolog may come back and redo this one …

Backtracking and Beads

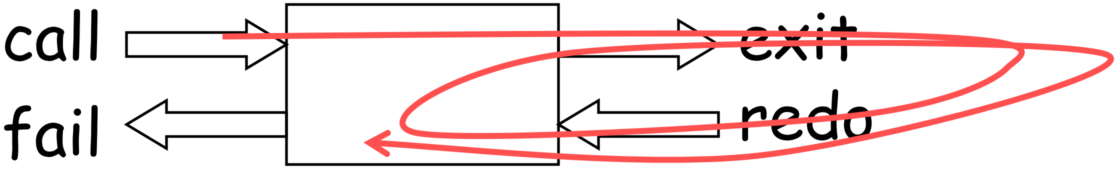
- Each Prolog constraint is like a “bead” in a string of beads:



we satisfy this constraint, making additional assignments, and move on …
but if our assignments cause later constraints to fail, Prolog may come back and redo this one …
let’s say we do find a new way to satisfy it.

Backtracking and Beads

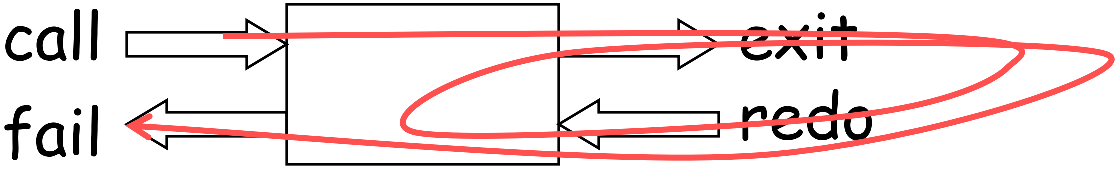
- Each Prolog constraint is like a “bead” in a string of beads:



If the new way still causes later constraints to fail, Prolog comes back through the redo port to try yet again.

Backtracking and Beads

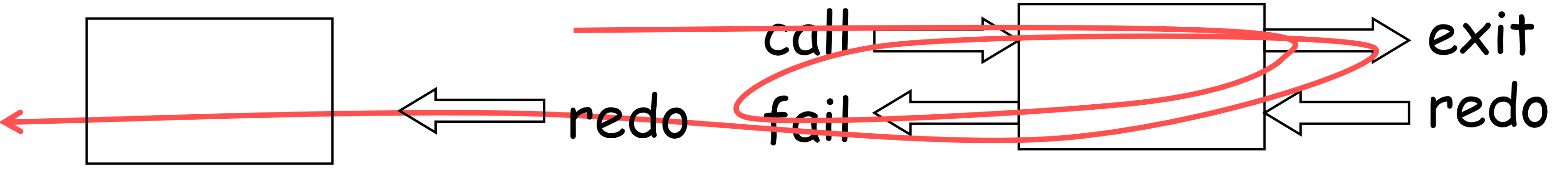
- Each Prolog constraint is like a "bead" in a string of beads:



If the new way still causes later constraints to fail, Prolog comes back through the redo port to try yet again.
If we're now out of solutions, we fail too …

Backtracking and Beads

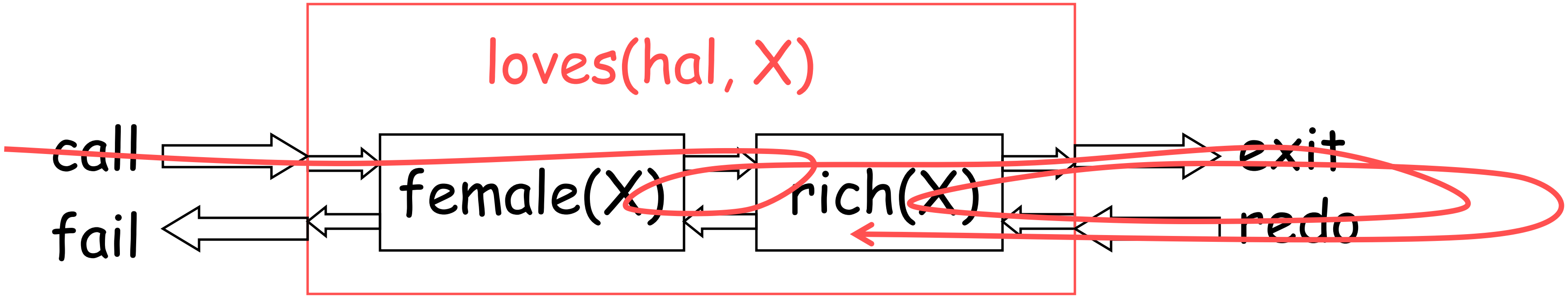
- Each Prolog constraint is like a "bead" in a string of beads:



If the new way still causes later constraints to fail, Prolog comes back through the redo port to try yet again.
If we're now out of solutions, we fail too … sending Prolog back to redo previous constraint.

Rules as nested beads

loves(hal, X) :- female(X), rich(X).

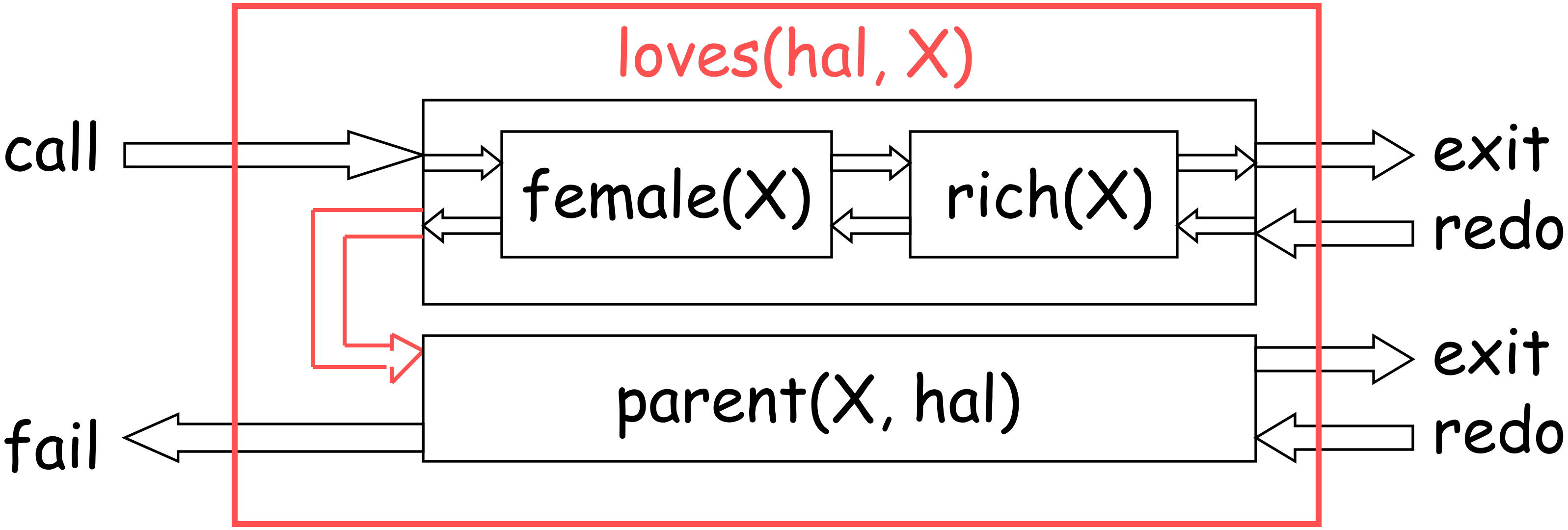


this is why you can backtrack into loves(hal,X)

slide thanks to David Matuszek (modified)

Alternative rules

```
loves(hal, X) :- female(X), rich(X).
loves(Child, X) :- parent(X, Child).
```

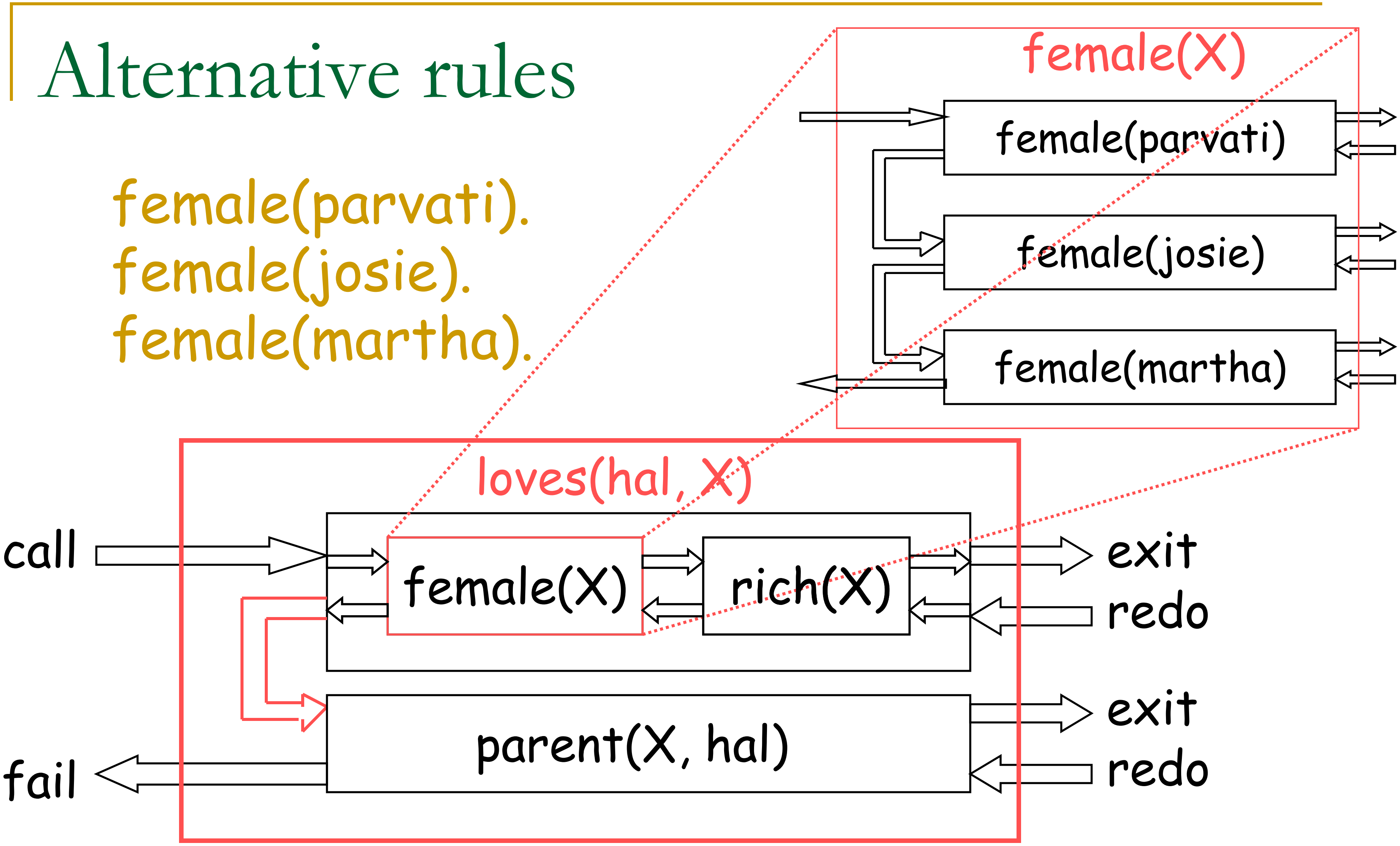


after running out of rich women, hal tries his parents

slide thanks to David Matuszek (modified)

Alternative rules

female(parvati).
female(josie).
female(martha).



slide thanks to David Matuszek (modified)

Prolog as a database language

- The various eats(…, …) facts can be regarded as rows in a database (2-column database in this case).
- Standard relational database operations:

```
•  eats(X,dal).                                   % select
•  edible(Object) :- eats(Someone, Object).       % project
•  parent(X,Y) :- mother(X,Y).                    % union
   parent(X,Y) :- father(X,Y).
•  sister_in_law(X,Z) :- sister(X,Y), married(Y,Z).   % join
```

- Why the heck does anyone still use SQL? Beats me.
- Warning: Prolog’s backtracking strategy can be inefficient.
  - But we can keep the little language illustrated above (“Datalog”) and instead compile into optimized query plans, just as for SQL.

Recursive queries

- Prolog allows recursive queries (SQL doesn't).
- Who's married to their boss?
  - boss(X,Y), married(X,Y).
- Who's married to their boss's boss?
  - boss(X,Y), boss(Y,Z), married(X,Z).
- Who's married to their boss's boss's boss?
  - Okay, this is getting silly. Let's do the general case.
- Who's married to someone above them?
  - above(X,X).
  - above(X,Y) :- boss(X,Underling), above(Underling,Y).
  - above(X,Y), married(X,Y).

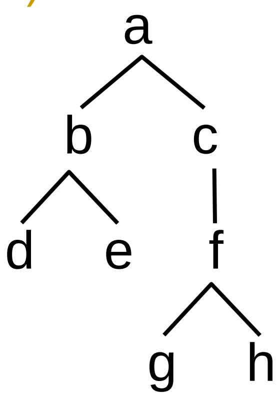
Base case. For simplicity, it says that any X is "above" herself.
If you don't like that, replace base case with above(X,Y) :- boss(X,Y).

Recursive queries

- **above(X,X).**
- above(X,Y) :- boss(X,Underling), above(Underling,Y).

- above(c,h).   % should return Yes
  - matches above(X,X)? no

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e).   …

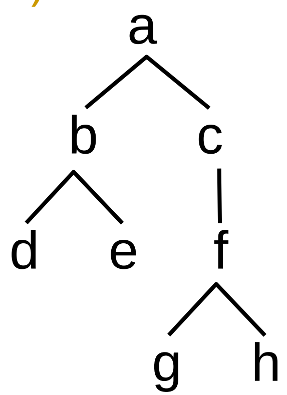


Recursive queries

- above(X,X).
- **above(X,Y) :- boss(X,Underling), above(Underling,Y).**

- above(c,h).   % should return Yes
  - matches above(X,Y) with X=c, Y=h
    - boss(c,Underling),
      - matches boss(c,f) with Underling=f
    - above(f, h).
      - matches above(X,X)? no

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e). …

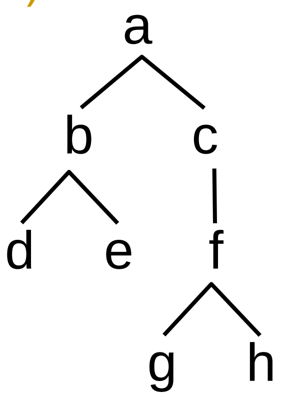


Recursive queries

- above(X,X).
- above(X,Y) :- boss(X,Underling), above(Underling,Y).

- above(c,h).   % should return Yes
  - matches above(X,Y) with X=c, Y=h
    - boss(c,Underling),
      - matches boss(c,f) with Underling=f
    - above(f, h).
      - matches above(X,Y) with X=f, Y=h
        (local copies of X,Y distinct from previous call)
        - boss(f,Underling),
        - matches boss(f,g) with Underling=g
        - above(g, h).
        - …ultimately fails because g has no underlings …

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e). …

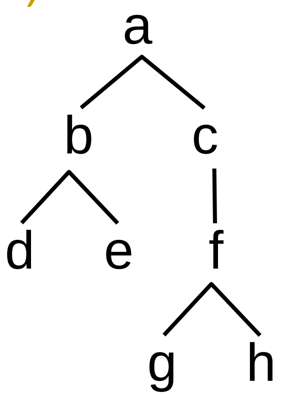


Recursive queries

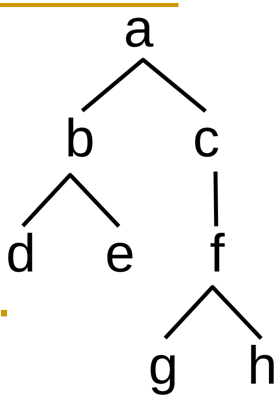
- above(X,X).
- above(X,Y) :- boss(X,Underling), above(Underling,Y).

- above(c,h).   % should return Yes
  - matches above(X,Y) with X=c, Y=h
    - boss(c,Underling),
      - matches boss(c,f) with Underling=f
    - above(f, h).
      - matches above(X,Y) with X=f, Y=h (local copies of X,Y distinct from previous call)
        - boss(f,Underling),
        - matches boss(f,h) with Underling=h
        - above(h, h).
        - matches above(X,X) with X=h

boss(a,b). boss(a,c).
boss(b,d). boss(c,f).
boss(b,e). …



Ordering constraints for speed



- □ above(X,X).
- □ above(X,Y) :- boss(X,Underling), above(Underling,Y).

- □ Which is more efficient?
- □ above(c,h), friends(c,h).
- □ friends(c,h), above(c,h).

Probably quicker to check first whether they're friends. If they're not, can skip the whole long above(c,h) computation, which must iterate through descendants of c.

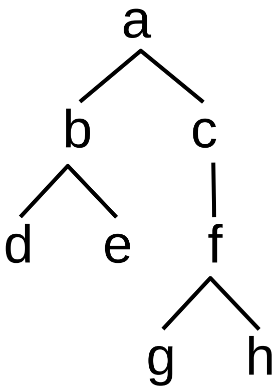
Which is more efficient?

above(X,Y), friends(X,Y).

friends(X,Y), above(X,Y).

For each boss X, iterate through all Y below her and check if each Y is her friend. (Worse to start by iterating through all friendships: if X has 5 friends Y, we scan all the people below her 5 times, looking for each friend in turn.)

Ordering constraints for speed



- above(X,X).

- Which is more efficient?

“query modes”

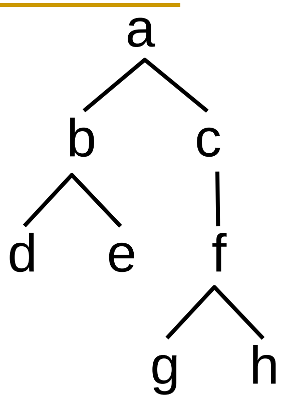
1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- boss(Overling,Y), above(X,Overling).

+,+ - If the query is above(c,e)?

  1. iterates over descendants of c, looking for e
  2. iterates over ancestors of e, looking for c.
  2. is better: no node has very many ancestors, but some have a lot of descendants.

+,- - If the query is above(c,Y)? 1. is better. Why?
-,+ - If the query is above(X,e)? 2. is better. Why?
-,- - If the query is above(X,Y)? Doesn't matter much. Why?

Ordering constraints for speed



- above(X,X).

- Which is more efficient?

“query modes“

1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- boss(Overling,Y), above(X,Overling).

+,+ □ If the query is above(c,e)?

1. iterates over descendants of c, looking for e
2. iterates over ancestors of e, looking for c.
2. is better: no node has very many ancestors, but some have a lot of descendants.

+,- □ If the query is above(c,Y)? 1. is better. Why?

-,+ □ If the query is above(X,e)? 2. is better. Why?

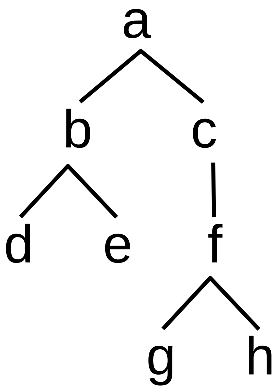
-,- □ If the query is above(X,Y)? Doesn't matter much. Why?

Warning: Actually, 1. has a significant advantage in Prolog implementations that do "1st-argument indexing."

That makes it much faster to find
a given x's children (boss(x,Y))
than a given y's parents (boss(X,y)).
So it is much faster to find descendants than ancestors.

If you don't like that, figure out how to tell your Prolog to do 2nd-argument indexing. Or just use subordinate(Y,X) instead of boss(X,Y)!

Ordering constraints for speed



- above(X,X).

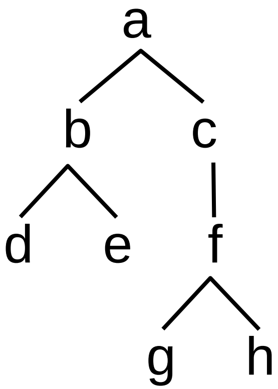
- Which is more efficient?

1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- above(Underling,Y), boss(X,Underling).

2. takes forever – literally!! Infinite recursion.

```
above(c,h).     % should return Yes
    matches above(X,Y) with X=c, Y=h
        above(Underling, h)
            matches above(X,Y) with X=Underling, Y=h
                above(Underling, h)
                    …
```

Ordering constraints for speed



- above(X,X).

- Which is more efficient?

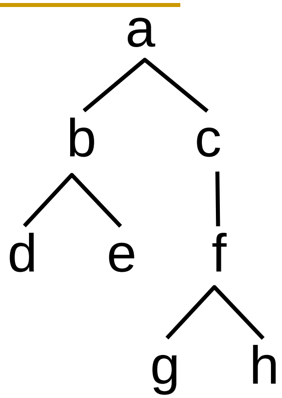
1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- above(Underling,Y), boss(X,Underling).

2. takes forever – literally!! Infinite recursion. Here's how:

above(c,h). % should return Yes

matches above(X,X)? no

Ordering constraints for speed



- above(X,X).

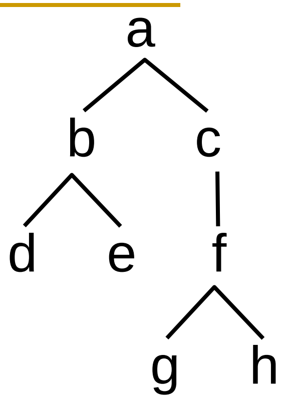
- Which is more efficient?

1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- above(Underling,Y), boss(X,Underling).

2. takes forever – literally!! Infinite recursion.
Here's how:

```
above(c,h).     % should return Yes
    matches above(X,Y) with X=c, Y=h
        above(Underling, h)
            matches above(X,X) with local X = Underling = h
        boss(c, h)    (our current instantiation of boss(X, Underling))
            no match
```

Ordering constraints for speed



- above(X,X).

- Which is more efficient?

1. above(X,Y) :- boss(X,Underling), above(Underling,Y).
2. above(X,Y) :- above(Underling,Y), boss(X,Underling).

2. takes forever – literally!! Infinite recursion.
Here's how:

```
above(c,h).     % should return Yes
     matches above(X,Y) with X=c, Y=h
          above(Underling, h)
               matches above(X,Y) with X=Underling, Y=h
                    above(Underling, h),
                         …
```

Prolog also allows complex terms

- What we’ve seen so far is called Datalog: “databases in logic.”
- Prolog is “programming in logic.” It goes a little bit further by allowing complex terms, including records, lists and trees.
- These complex terms are the source of the only hard thing about Prolog, “unification.”

Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- Several essentially identical ways to find older students:
- at_jhu(student(IDNum, Name, date(Day,Month,Year))),
  Year < 1983.
- at_jhu(student(_, Name, date(_,_,Year))),
  Year < 1983.
- at_jhu(Person),
  Person=student(_,_,Birthday),
  Birthday=date(_,_,Year),
  Year < 1983.

usually no need to use =
but sometimes it’s nice
to introduce a temporary name
especially if you’ll use it twice

This query binds Person and Birthday to complex structured values, and Year to an int. Prolog prints them all.

example adapted from Ian Davey-Wilson

```
homepage(html(head(title("Peter A. Flach")),
              body([img([align=right,src="logo.jpg"]),
                    img([align=left,src="peter.jpg"]),
                    h1("Peter Flach's homepage"),
                    h2("Research interests"),
                    ul([li("Learning from structured data"),
                        ...,
                        li(a([href="CV.pdf"],"Full CV"))]),
                    h2("Current activities"),
                    ...,
                    h2("Past activities"),
                    ...,
                    h2("Archives"),
                    ...,
                    hr,address(…)
                  ])
     )).
```

One big term representing an HTML web page.

The style on the previous slide could get unmanageable.

You have to remember that birthday is argument #3 of person, etc.

This nondeterministic query asks whether the page title is a person and "Research" appears in some heading on the page.

```
pagetype(Webpage,researcher):-
    page_get_head(Webpage,Head),
    head_get_title(Head, Title),
    person(Title),
    page_get_body(Webpage,Body),
    body_get_heading(Body,Heading),
    substring("Research",Heading).
```

slide thanks to Peter A. Flach (modified)

Complex terms

- at_jhu(student(128327, ‘Spammy K',   date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’,   date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W',   date(23, aug, 1966))).

- ~~student_get_bday(Stu , Bday) :- Stu=student(_, _, Bday).~~
- ~~date_get_year(Date,Year) :- Date=date(_, _, Year).~~ bad style

- So you could write accessors in object-oriented style:
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.
- Answer:
  Student=student(456591, ‘Fuzzy W’, date(23, aug, 1966)),
  Birthday=date(23, aug, 1966),
  Year=1966.

Complex terms

- at_jhu(student(128327, ‘Spammy K',   date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’,   date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W',   date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday),   Bday)   .
- date_get_year(date(_, _, Year), Year).

good style

- So you could write accessors in object-oriented style:
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.
- Answer:
  Student=student(456591, ‘Fuzzy W’, date(
  Birthday=date(23, aug, 1966),
  Year=1966.

whoa, what are the variable bindings at this point?? Student&Birthday weren’t forced to particular values by the constraint. But were forced into a relation …

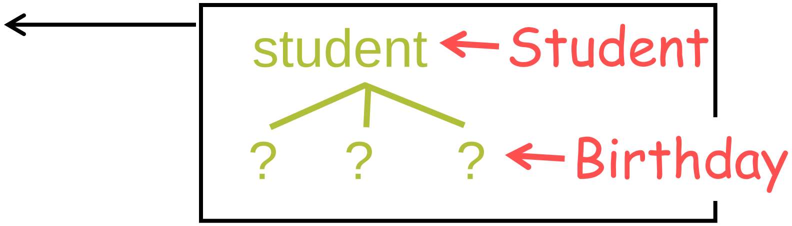
Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday) .
- date_get_year(date(_, _, Year), Year).

good style

- So you could write accessors in object-oriented style:
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.



- Answer:
  Student=student(456591, ‘Fuzzy W’, date(23, aug, 1966)),
  Birthday=date(23, aug, 1966),
  Year=1966.

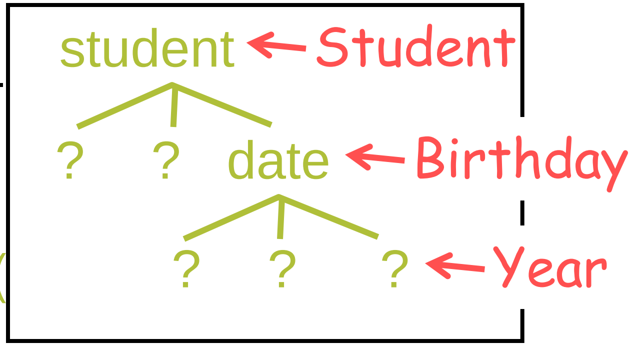
Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’,   date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W',   date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday),   Bday)   .
- date_get_year(date(_, _, Year), Year).

good style

- So you could write accessors in object-oriented style:
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.
- Answer:
  Student=student(456591, ‘Fuzzy W’, date(
  Birthday=date(23, aug, 1966),
  Year=1966.



Complex terms

```
at_jhu(student(128327, ‘Spammy K',  date(2, may, 1986))).
at_jhu(student(126547, ‘Blobby B’,    date(15, dec, 1985))).
at_jhu(student(456591, ‘Fuzzy W',     date(23, aug, 1966))).

student_get_bday(student(_, _, Bday),      Bday)           .
date_get_year(date(_, _, Year), Year).
```

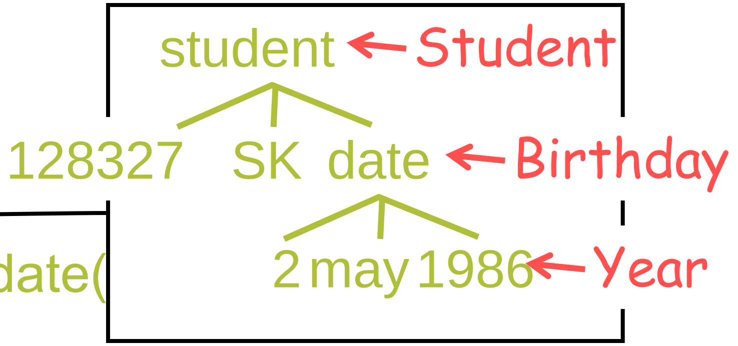
good style

- So you could write accessors in object-oriented style:

```
student_get_bday(Student,Birthday),
    date_get_year(Birthday,Year),
    at_jhu(Student), Year < 1983.
```

- Answer:

```
    Student=student(456591, ‘Fuzzy W’, date(
    Birthday=date(23, aug, 1966),
    Year=1966.
```



Complex terms

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday) .
- date_get_year(date(_, _, Year), Year).

good style

- So you could write accessors in object-
- student_get_bday(Student,Birthday),
  date_get_year(Birthday,Year),
  at_jhu(Student), Year < 1983.
- Answer:
  Student=student(456591, ‘Fuzzy W’, date(
  Birthday=date(23, aug, 1966),
  Year=1966.

Fail

(and backtrack)

How does matching happen?

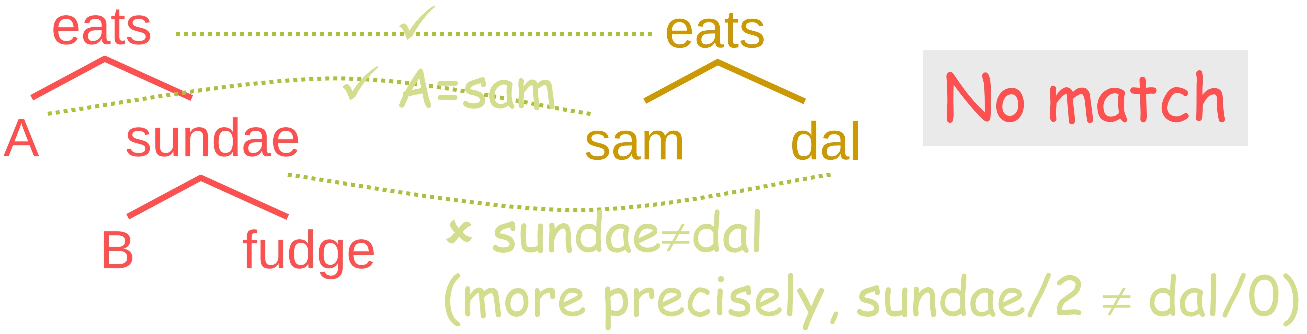
- eats(sam, dal).
- eats(josie, sundae(vanilla, caramel)).
- eats(rajiv, sundae(mintchip, fudge)).
- eats(robot(’C-3PO’), Anything). % variable in a fact

- Query: eats(A, sundae(B,fudge)).
- Answer: A=rajiv, B=mintchip

How does matching happen?

- **eats(sam, dal).**
- eats(josie, sundae(vanilla, caramel)).
- eats(rajiv, sundae(mintchip, fudge)).
- eats(robot('C-3PO'), Anything). % variable in a fact

- Query: eats(A, sundae(B,fudge)).
- What happens when we try to match this against facts?



No match

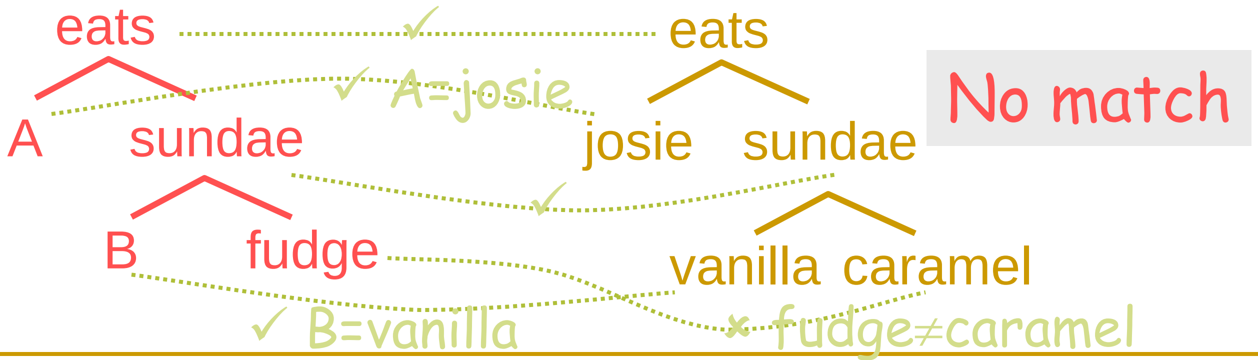
× sundae≠dal

(more precisely, sundae/2 ≠ dal/0)

How does matching happen?

- eats(sam, dal).
- **eats(josie, sundae(vanilla, caramel)).**
- eats(rajiv, sundae(mintchip, fudge)).
- eats(robot('C-3PO'), Anything). % variable in a fact

- Query: eats(A, sundae(B,fudge)).
- What happens when we try to match this against facts?



eats ✓ eats

A sundae

✓ A=josie

josie sundae

✓

B fudge

vanilla caramel

✓ B=vanilla

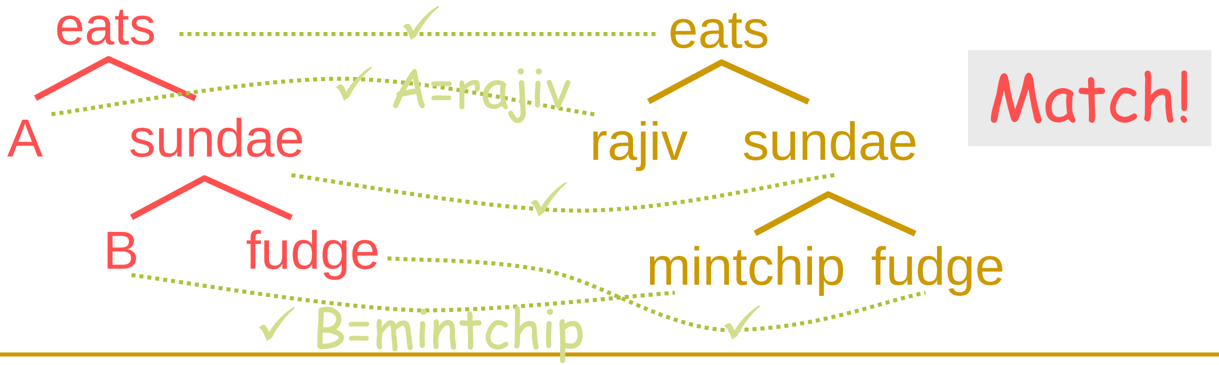
✗ fudge≠caramel

No match

How does matching happen?

- eats(sam, dal).
- eats(josie, sundae(vanilla, caramel)).
- **eats(rajiv, sundae(mintchip, fudge)).**
- eats(robot('C-3PO'), Anything). % variable in a fact

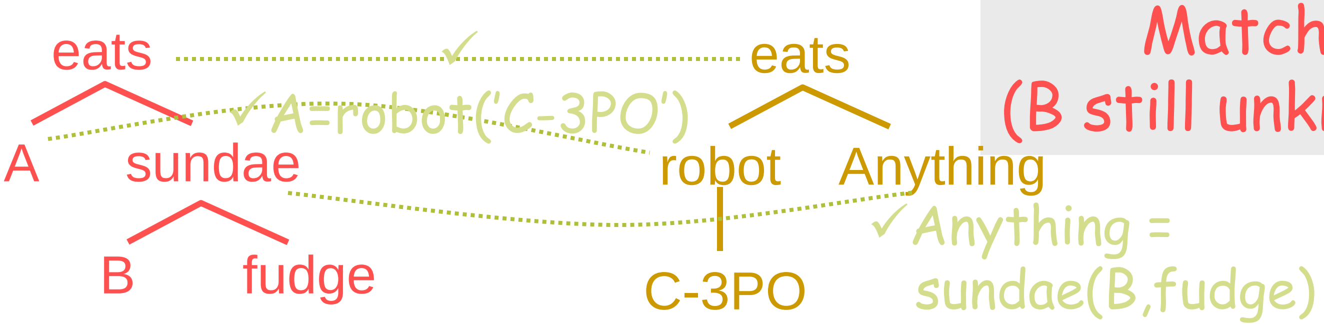
- Query: eats(A, sundae(B,fudge)).
- What happens when we try to match this against facts?



How does matching happen?

- eats(sam, dal).
- eats(josie, sundae(vanilla, caramel)).
- eats(rajiv, sundae(mintchip, fudge)).
- **eats(robot('C-3PO'), Anything).** % variable in a fact

- Query: eats(A, sundae(B,fudge)), icecream(B).
- What happens when we try to match this against facts?



Match!
(B still unknown)

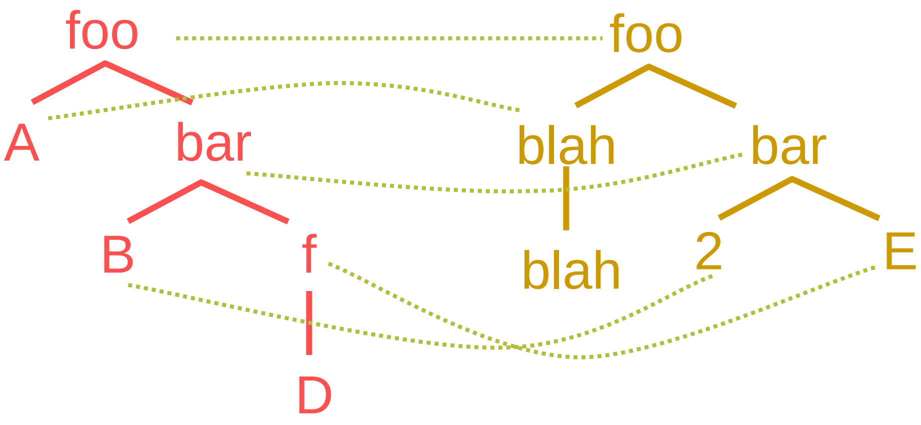
How does matching happen?

- eats(sam, dal).
- eats(josie, sundae(vanilla, caramel)).
- eats(rajiv, sundae(mintchip, fudge)).
- **eats(robot(’C-3PO’), Something) :- food(Something).**
- food(dal).   icecream(vanilla).
- food(fudge).   icecream(chocolate).
- **food(sundae(Base, Topping)) :- icecream(Base),**
  food(Topping).

- Query: eats(robot(A), sundae(B,fudge)).
- Answer: A=’C-3PO’, B can be any kind of ice cream

How does matching happen?

- Let’s use a “=” constraint to invoke unification directly …
- Query: foo(A,bar(B,f(D))) = foo(blah(blah), bar(2,E)).
- Answer: A=blah(blah), B=2, E=f(D)



This is like unit propagation in DPLL SAT solvers.

- Unifying 2 nodes “propagates”: it forces their children to be unified too.
  (As in DPLL, propagation could happen in any order. Options?)
- This may bind some unassigned variables to particular nodes.
  (Like assigning A=0 or A=1 in DPLL.)
- In case of a conflict, backtrack to prev. decision, undoing all propagation.

Two obvious recursive definitions

- **Term** (the central data structure in Prolog programs)
  1. Any variable is a term (e.g., X).
  2. Any atom (e.g., foo) or other simple constant (e.g., 7) is a term.
  3. If f is an atom and t_1, t_2, … t_n are terms,
     then f(t_1, t_2, … t_n) is a term.

  This lets us build up terms of any finite depth.

- **Unification** (matching of two terms $\alpha=\beta$)
  1. If α or β is a variable, $\alpha=\beta$ succeeds and returns immediately: side effect is to bind that variable.
  2. If α is f(t_1, t_2, … t_n) and β is f(t_1', t_2', … t_n'), then recurse:
     $\alpha=\beta$ succeeds iff we can unify children $t_1=t_1'$, $t_2=t_2'$, … $t_n=t_n'$.

     n=0 is the case where α, β are atoms or simple constants.
  3. In all other cases, $\alpha=\beta$ fails (i.e., conflict).

Two obvious recursive definitions

More properly, if it's still unknown ("?"), given bindings so far.

Consider foo(X,X)=foo(3,7). Recurse:

- First we unify X=3. Now X is no longer unknown.
- Then try to unify X=7, but since X already bound to 3, this tries to unify 3=7 and fails. X can't be both 3 and 7.

(Like the conflict from assigning X=0 and then X=1 during DPLL propagation.)

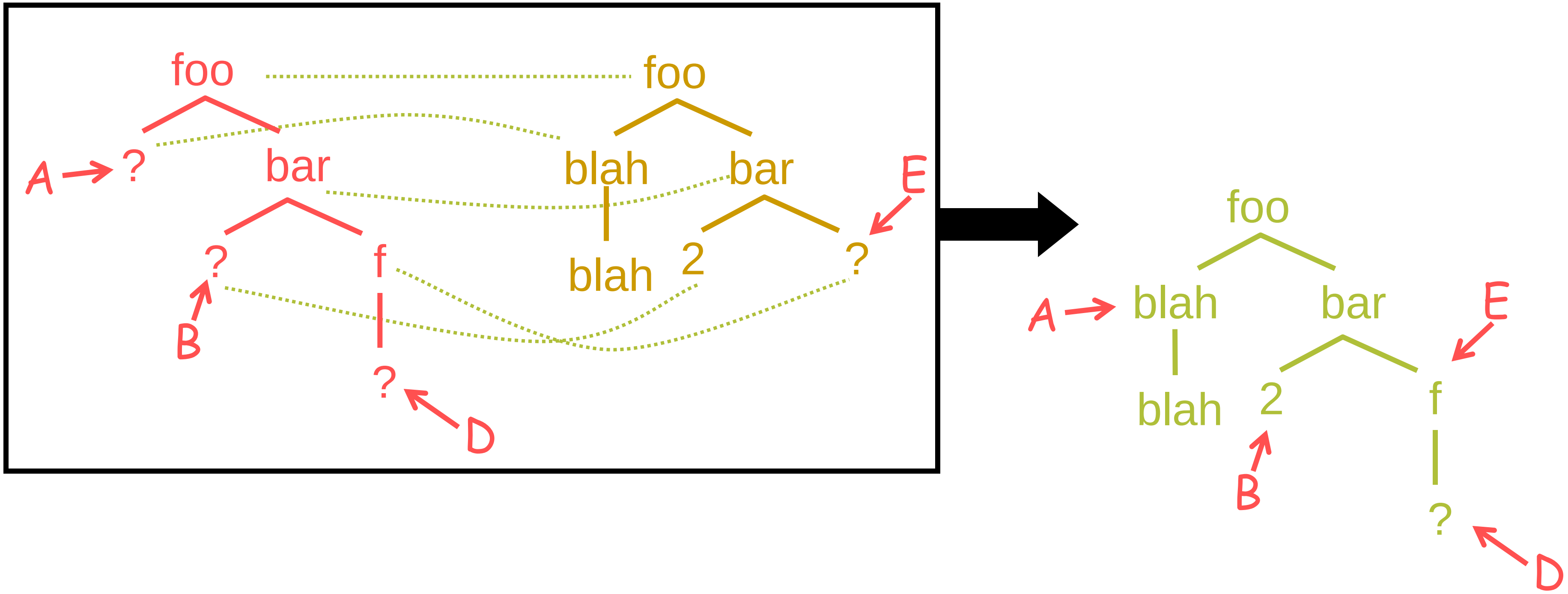
How about: foo(X1,X2)=foo(3,7), X1=X2? Or X1=X2, foo(X1,X2)=foo(3,7)?

- **Unification** (matching of two terms $\alpha=\beta$)
  1. If α or β is a variable, $\alpha=\beta$ succeeds and returns immediately: side effect is to bind that variable.
  2. If α is $f(t_1, t_2, \ldots t_n)$ and β is $f(t_1', t_2', \ldots t_n')$, then recurse: $\alpha=\beta$ succeeds iff we can unify children $t_1=t_1'$, $t_2=t_2'$, … $t_n=t_n'$.

     n=0 is the case where α, β are atoms or simple constants.
  3. In all other cases, $\alpha=\beta$ fails (i.e., conflict).

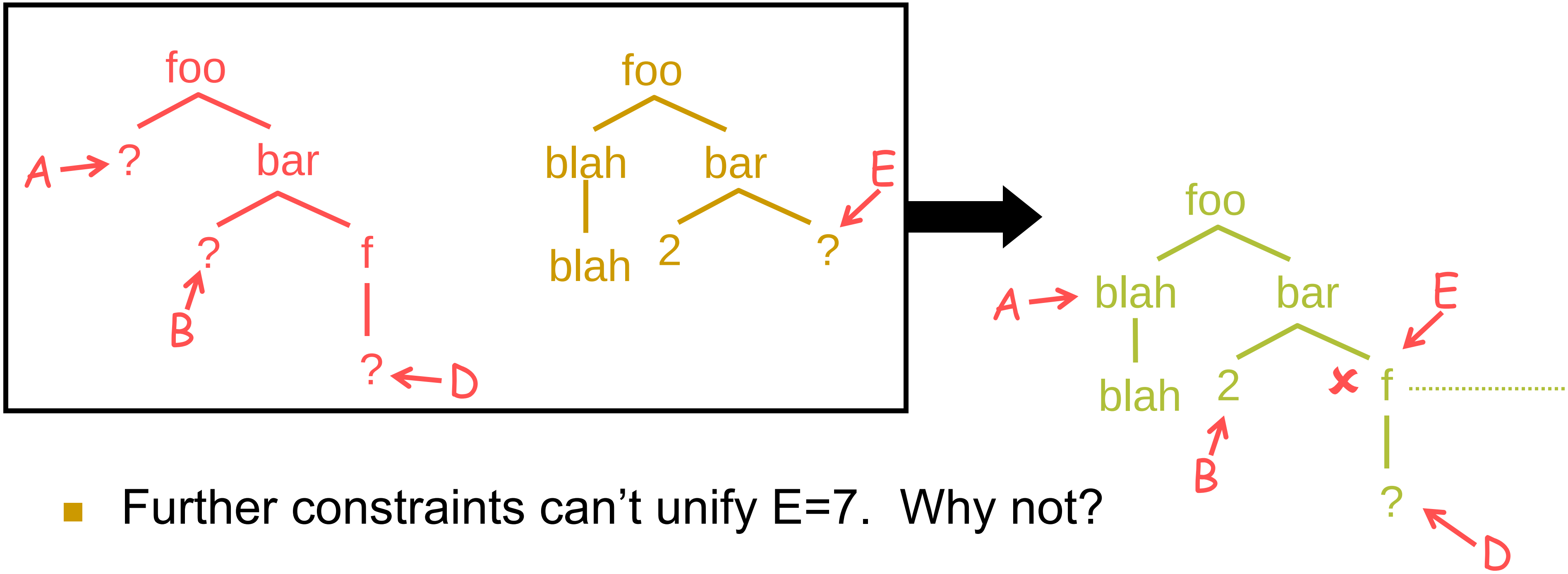
Variable bindings resulting from unification

- Let’s use the “=” constraint to invoke unification directly …
- Query: foo(A,bar(B,f(D))) = foo(blah(blah), bar(2,E)).
- Answer: A=blah(blah), B=2, f(D)=E



Variable bindings resulting from unification

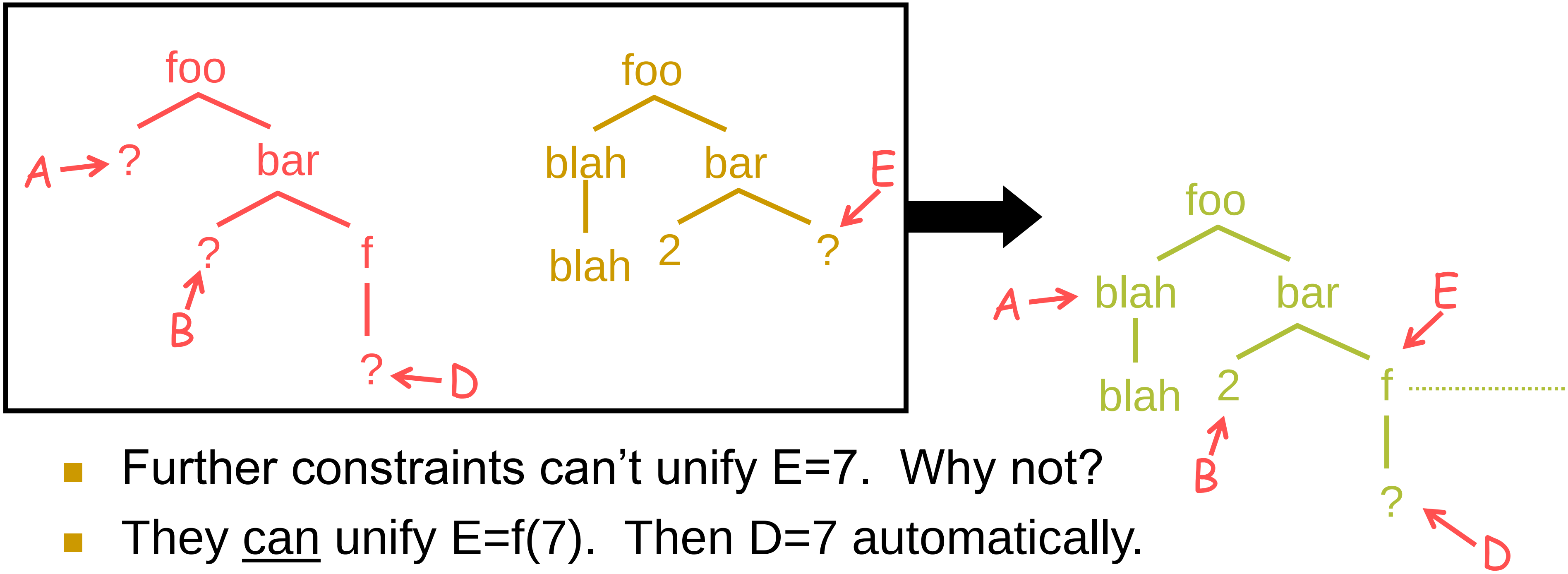
- The “=” constraint invokes unification directly …
- Query: foo(A,bar(B,f(D))) = foo(blah(blah), bar(2,E)).
- Answer: A=blah(blah), B=2, f(D)=E



- Further constraints can’t unify E=7. Why not?

Variable bindings resulting from unification

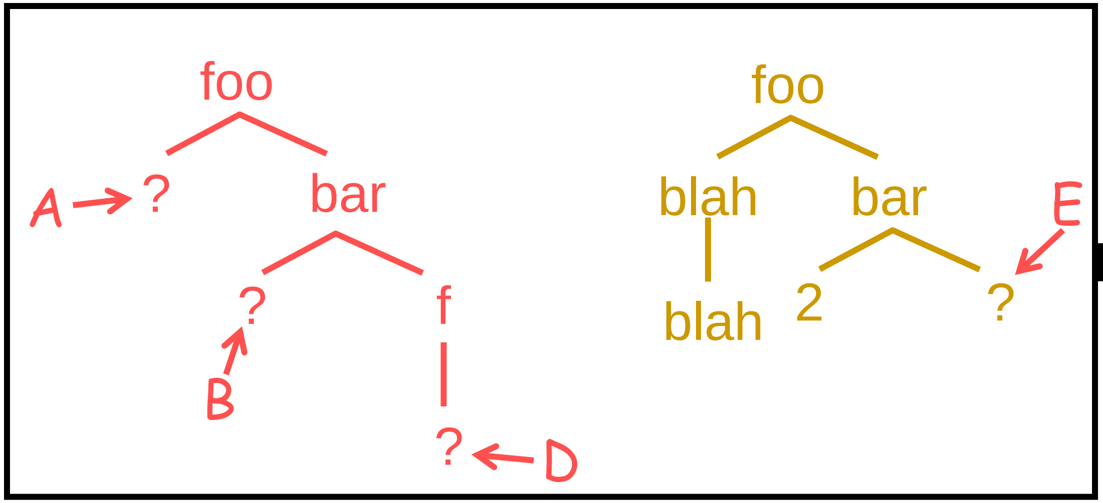
- The “=” constraint invokes unification directly …
- Query: foo(A,bar(B,f(D))) = foo(blah(blah), bar(2,E)).
- Answer: A=blah(blah), B=2, f(D)=E



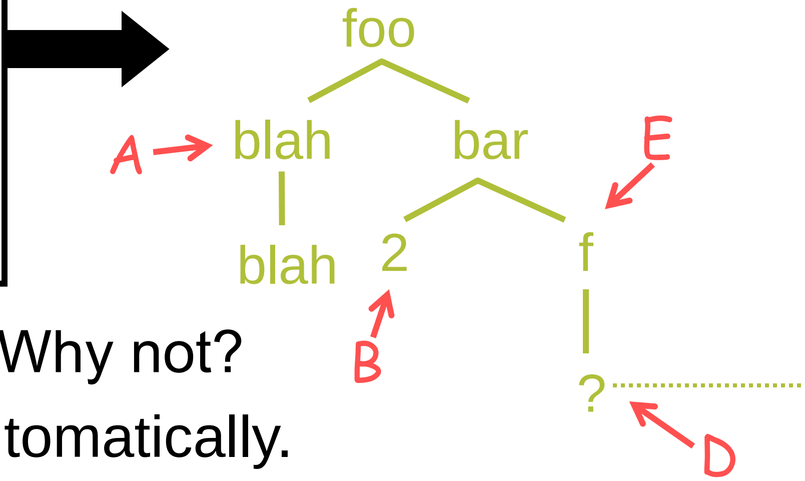
- Further constraints can’t unify E=7. Why not?
- They can unify E=f(7). Then D=7 automatically.

Variable bindings resulting from unification

- The "=" constraint invokes unification directly …
- Query: foo(A,bar(B,f(D))) = foo(blah(blah), bar(2,E)).
- Answer: A=blah(blah), B=2, f(D)=E



Note: All unification is undone upon backtracking!



- Further constraints can't unify E=7. Why not?
- They can unify E=f(7). Then D=7 automatically.
- Or if they unify D=7, then E=f(7) automatically.

Two obvious recursive definitions

Even X=f(X) succeeds, with X=the weird <u>circular</u> term f(f(f(…))).
Our definitions of terms and unification don't allow circularity.
So arguably X=f(X) should just fail. Unsatisfiable constraint!
But this "occurs check" would be slow, so Prolog skips it.

- **Unification** (matching of two terms $\alpha=\beta$)
  1. If α or β is a variable, $\alpha=\beta$ succeeds and returns immediately: side effect is to bind that variable.
  2. If α is $f(t_1, t_2, \ldots t_n)$ and β is $f(t_1', t_2', \ldots t_n')$, then recurse: $\alpha=\beta$ succeeds iff we can unify children $t_1=t_1'$, $t_2=t_2'$, … $t_n=t_n'$.

     n=0 is the case where α, β are atoms or simple constants.
  3. In all other cases, $\alpha=\beta$ fails (i.e., conflict).

When does Prolog do unification?

1. To satisfy an "α=β" constraint.
2. To satisfy any other constraint α. Prolog tries to unify it with some β that is the head of a clause in your program:
   - β. % a fact
   - β :- γ_1, γ_2, γ_3. % a rule

- Prolog's decisions = which clause from your program to pick.
  - Like decision variables in DPLL, this is the nondeterministic choice part.
- A decision "propagates" in two ways:
  - Unifying nodes forces their children to unify, as we just saw.
    - Like unit propagation in DPLL. Can fail, forcing backtracking.
  - After unifying α=β where β is a rule head, we are forced to satisfy constraints γ_1, γ_2, γ_3 from the rule's body (requiring more unification).
    - How to satisfy them may involve further decisions, unlike DPLL.
- Variable bindings that arise during a unification may affect Prolog's ability to complete the unification, or to do subsequent unifications that are needed to satisfy additional constraints (e.g., those from clause body).
  - Bindings are undone upon backtracking, up to the last decision for which other options are available.

Note: The = constraint isn't really special

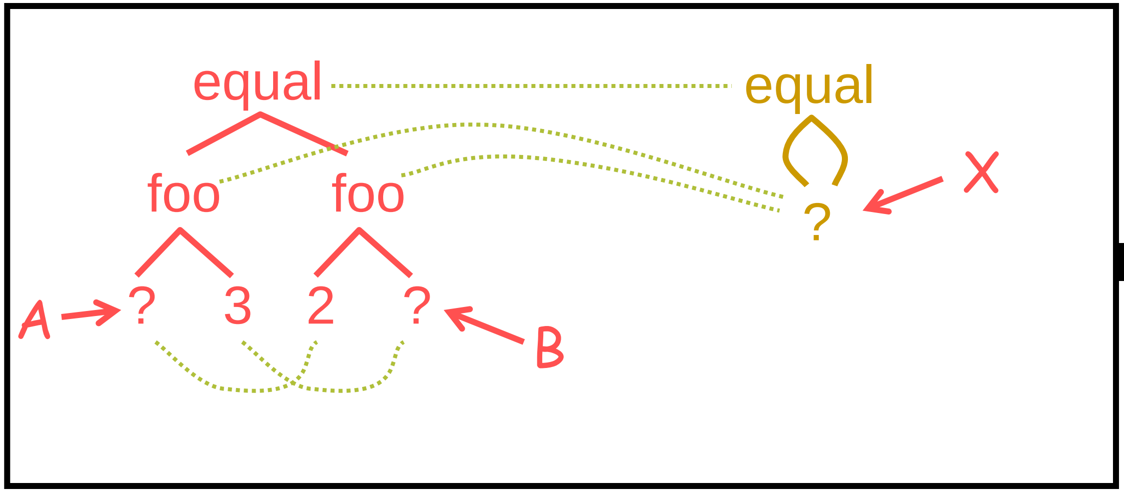
1. To process an "α=β" constraint.

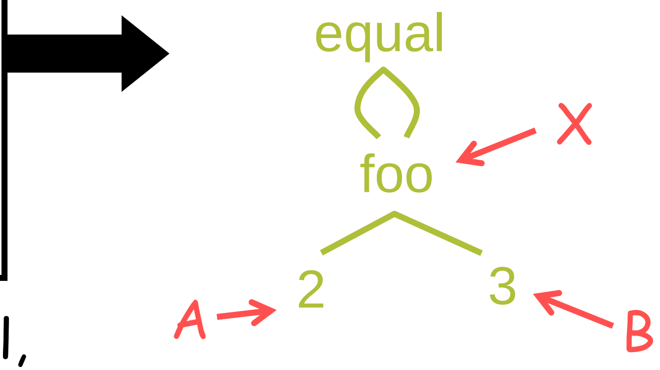
- Actually, this is not really special. You could implement = if it weren't built in. Just put this fact in your program:
  - equal(X,X).
- Now you can write the constraint
  - equal(foo(A,3), foo(2,B)).
- How would Prolog try to satisfy the constraint?
  - It would try to unify equal(X,X) with equal(foo(A,3), foo(2,B)).
  - This means unifying X with foo(A,3) and X with foo(2,B).
  - So foo(A,3) would indirectly get unified with foo(2,B), yielding A=2, B=3.

Note: The = constraint isn't really special

- Query: equal(foo(A,3), foo(2,B)).
- Unify against program fact: equal(X,X).

The unification wouldn't have succeeded if there hadn't been a way to instantiate A,B to make the foo terms equal.





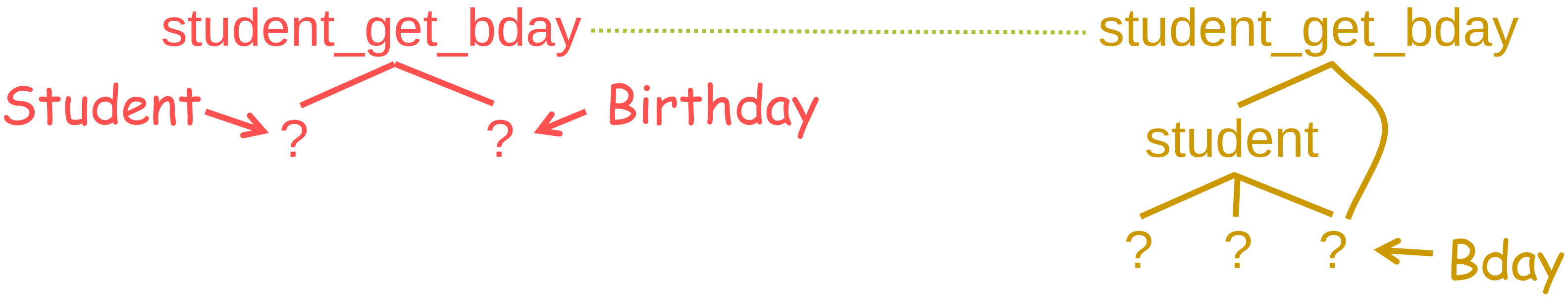
If we wanted to call it = instead of equal, we could write '='(X,X) as our program fact. Prolog even lets you declare '=' as infix, making X=X a synonym for '='(X,X).

Now we should really get the birthday example

- at_jhu(student(128327, ‘Spammy K',   date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’,   date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W',   date(23, aug, 1966))).

- **student_get_bday(student(_, _, Bday), Bday).**
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday),

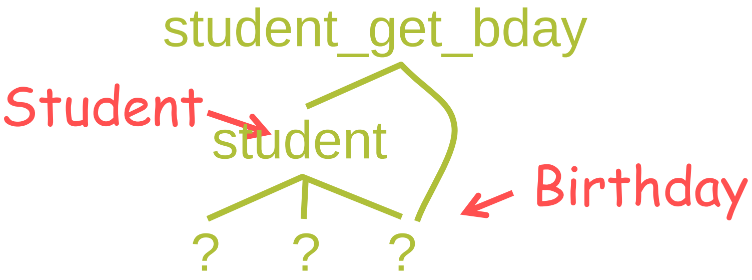


Now we should really get the birthday example

- at_jhu(student(128327, ‘Spammy K',   date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’,   date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W',   date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday),

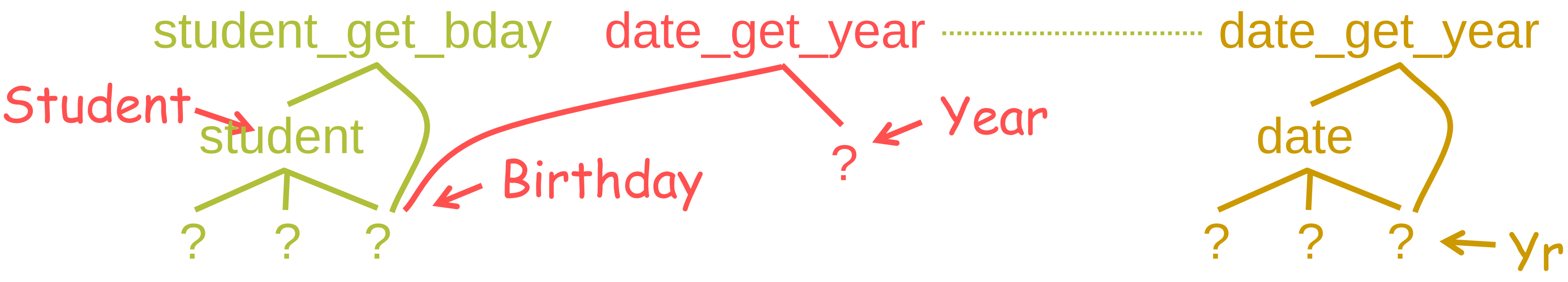


Now we should really get the birthday example

```
at_jhu(student(128327, 'Spammy K',   date(2, may, 1986))).
at_jhu(student(126547, 'Blobby B',   date(15, dec, 1985))).
at_jhu(student(456591, 'Fuzzy W',    date(23, aug, 1966))).

student_get_bday(student(_, _, Bday), Bday).
date_get_year(date(_, _, Yr), Yr).
```

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year),

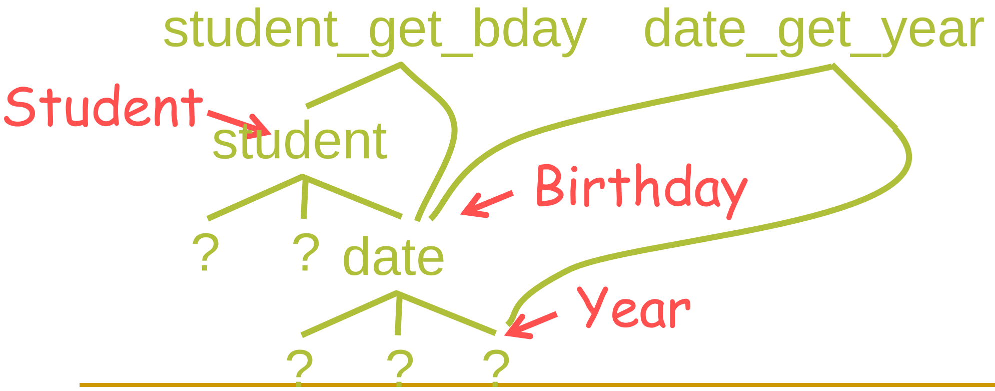


Now we should really get the birthday example

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year),

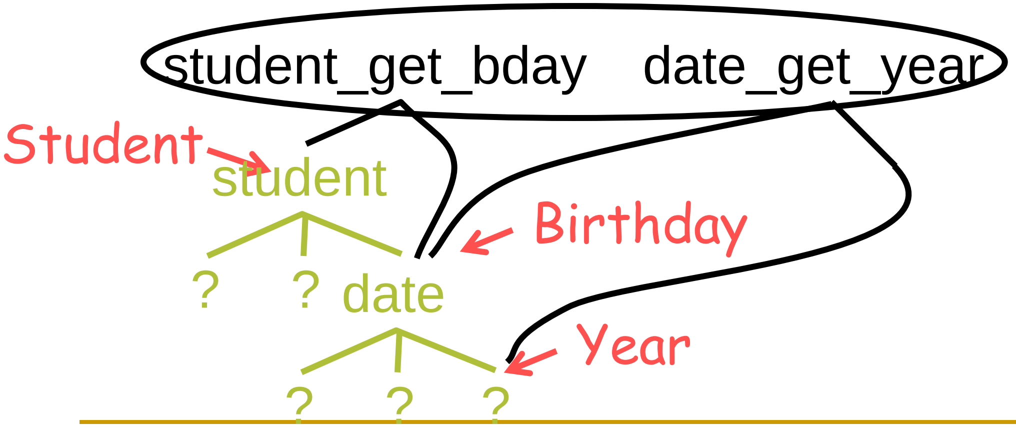


Now we should really get the birthday example

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year),



Note: We don’t really care about the black pieces anymore. They are just left-over junk that helped us satisfy previous constraints. We could even garbage-collect them now, since no variables point to them.

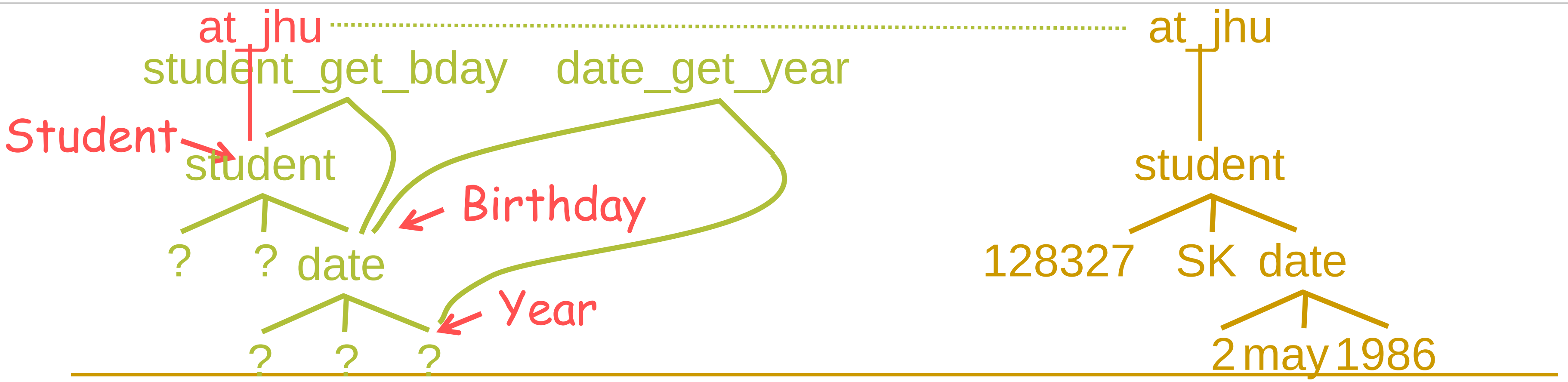
The rest of the structure is exactly what we hoped for (earlier slide).

Now we should really get the birthday example

- **at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).**
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year), at_jhu(Student),



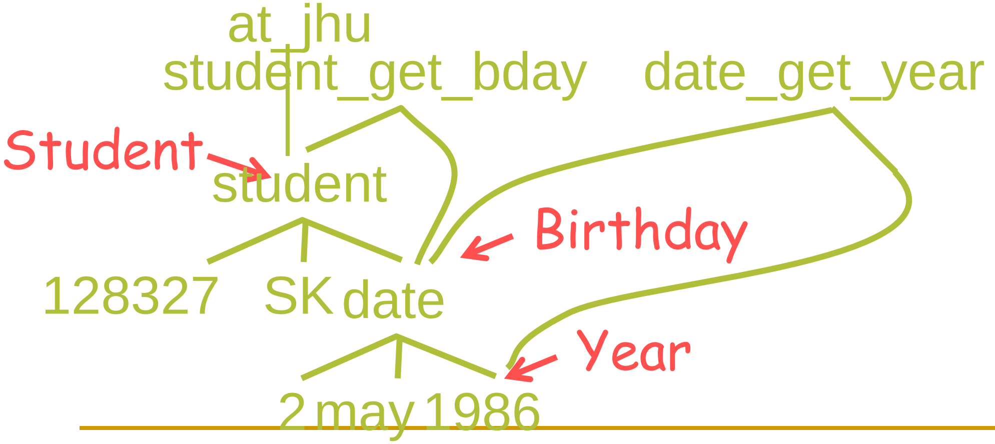
at_jhu
student_get_bday
date_get_year
student
? ? date
? ? ?
at_jhu
student
128327 SK date
2 may 1986

Now we should really get the birthday example

- at_jhu(student(128327, ‘Spammy K',   date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’,   date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W',   date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year), at_jhu(Student),

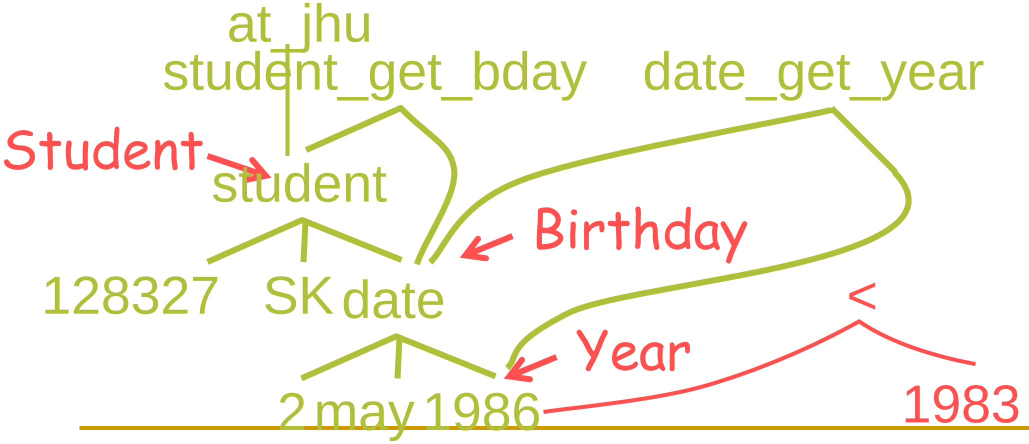


Now we should really get the birthday example

- at_jhu(student(128327, ‘Spammy K',   date(2, may, 1986))).
- at_jhu(student(126547, ‘Blobby B’,   date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W',   date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year), at_jhu(Student), Year < 1983.



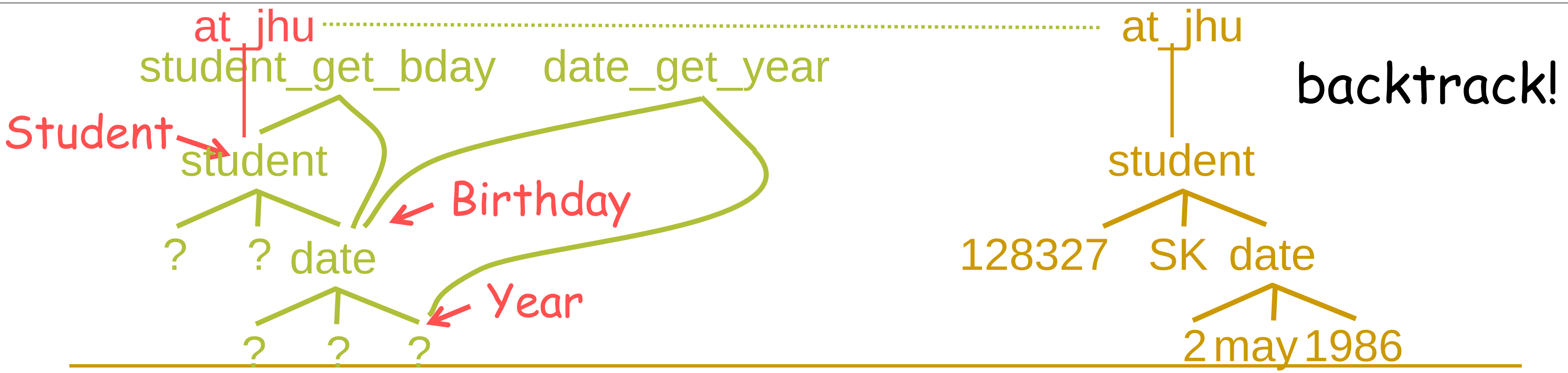
fail! 1986 < 1983 doesn’t match anything in database. (Well, okay, actually < is built-in.)

Now we should really get the birthday example

- **at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).**
- at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year), at_jhu(Student),

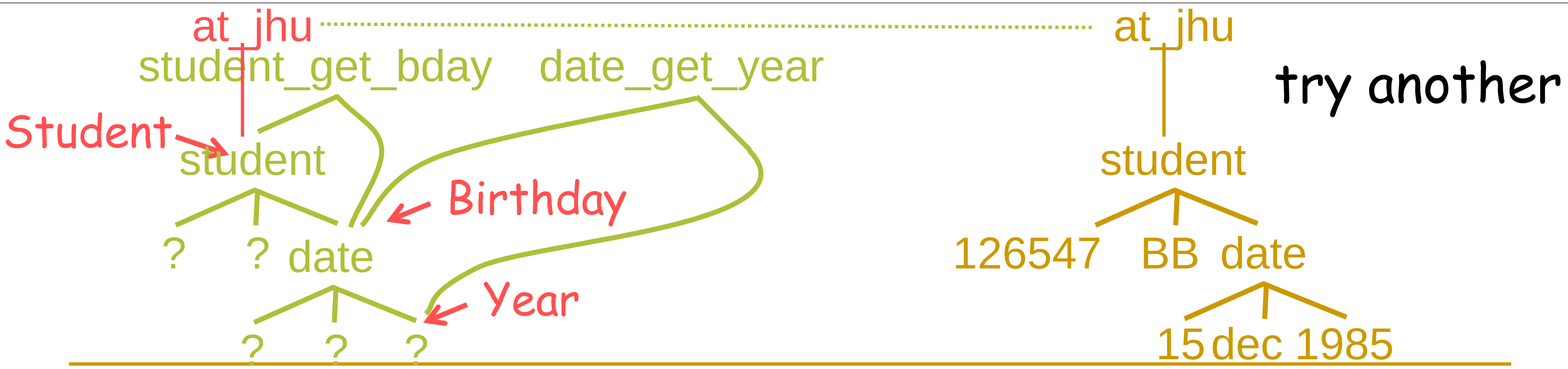


Now we should <u>really</u> get the birthday example

- at_jhu(student(128327, ‘Spammy K', date(2, may, 1986))).
- **at_jhu(student(126547, ‘Blobby B’, date(15, dec, 1985))).**
- at_jhu(student(456591, ‘Fuzzy W', date(23, aug, 1966))).

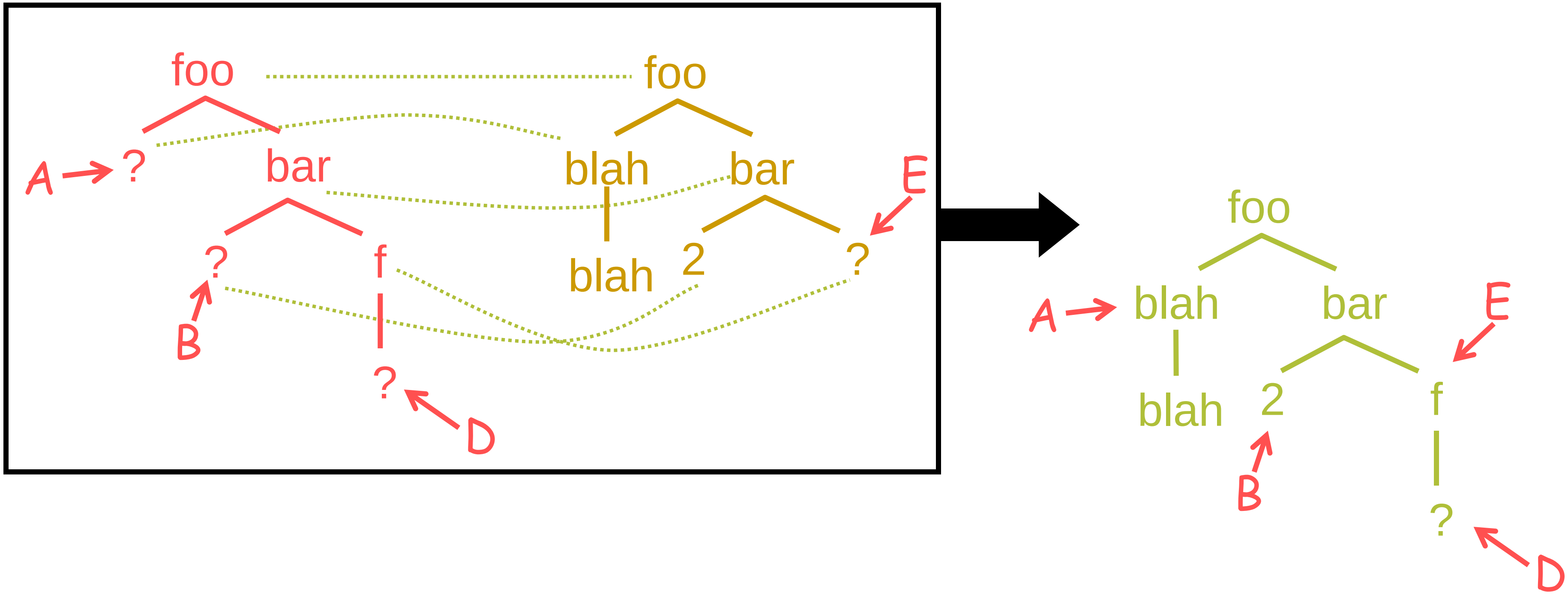
- student_get_bday(student(_, _, Bday), Bday).
- date_get_year(date(_, _, Yr), Yr).

- student_get_bday(Student,Birthday), date_get_year(Birthday,Year), at_jhu(Student),



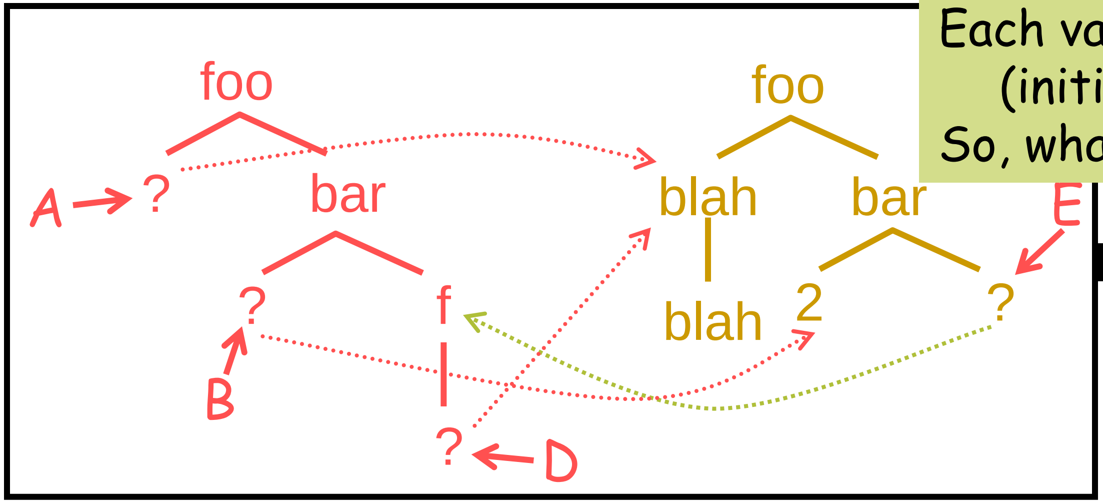
Variable bindings resulting from unification

- Let’s use the “=” constraint to invoke unification directly …
- Query: foo(A,bar(B,f(D))) = foo(blah(blah), bar(2,E)).
- Answer: A=blah(blah), B=2, f(D)=E

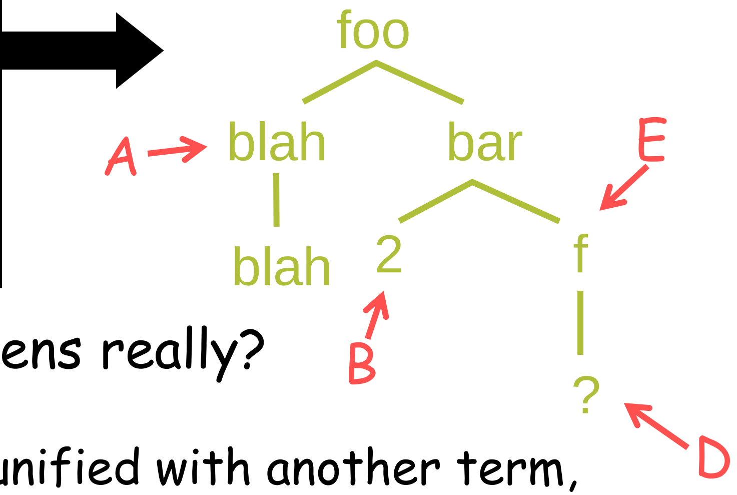


Variable bindings resulting from unification

- The “=” constraint invokes unification directly …
- Query: foo(A,bar(B,f(D))) = foo(blah(blah), bar(2,E)).
- Answer: A=blah(blah), B=2, f(D)=E



Each variable name stores a pointer too (initially to a new “?”).
So, what happens if we now unify A=D?



In memory, it’s not animated. ☺ What happens really?
Each ? stores a pointer.
Initially it’s the null pointer, but when ? is first unified with another term, change it to point to that term. (This is what’s undone upon backtracking.)
Future accesses to the ? don’t see the ?; they transparently follow its pointer.
(If two ?’s with null pointers are unified, pick one and make it point to the other (just as in the Union-Find algorithm). This may lead to chains of pointers.)

Time to try some programming!

- Now you know how the Prolog solver works. (It helps to know in advance.)
- Let’s try some programming!
- We’ll try recursion again, but this time with complex terms.

Family trees (just Datalog here) …

```
female(sarah).
female(rebekah).
female(hagar_concubine).
female(milcah).
female(bashemath).
female(mahalath).
female(first_daughter).
female(second_daughter).
female(terahs_first_wife).
female(terahs_second_wife).
female(harans_wife).
female(lots_first_wife).
female(ismaels_wife).
female(leah).
female(kemuels_wife).
female(rachel).
female(labans_wife).
```

```
male(terah).
male(nahor).
male(isaac).
male(uz).
male(bethuel).
male(iscah).
male(jacob).
male(hadad).
male(reuel).
male(judah4th).
male(elak).
male(ben-ammi).
```

```
male(abraham).
male(haran).
male(ismael).
male(kemuel).
male(lot).
male(esau).
male(massa).
male(laban).
male(levi3rd).
male(aliah).
male(moab).
```

Family trees (just Datalog here) …

```
father(terah, sarah).
father(terah, abraham).
father(terah, nahor).
father(terah, haran).
father(abraham, isaac).
father(abraham, ismael).
father(nahor, uz).
father(nahor, kemuel).
father(nahor, bethuel).
father(haran, milcah).
father(haran, lot).
father(haran, iscah).
father(isaac, esau).
father(isaac, jacob).
father(ismael, massa).
father(ismael, mahalath).
father(ismael, hadad).
father(ismael, bashemath).
father(esau, reuel).
father(jacob, levi3rd).
father(jacob, judah4th).
father(esau, aliah).
father(esau, elak).
father(kemuel, aram).
father(bethuel, laban).
father(bethuel, rebekah).
father(lot, first_daughter).
father(lot, second_daughter).
father(lot, moab).
father(lot, ben_ammi).
father(laban, rachel).
father(laban, leah).
```

```
mother(terahs_second_wife, sarah).
mother(terahs_first_wife, abraham).
mother(terahs_first_wife, nahor).
mother(terahs_first_wife, haran).
mother(sarah, isaac).
mother(hagar_concubine, ismael).
mother(milcah, uz).
mother(milcah, kemuel).
mother(milcah, bethuel).
mother(harans_wife, milcha).
mother(harans_wife, lot).
mother(harans_wife, iscah).
mother(rebekah, esau).
mother(rebekah, jacob).
mother(ismaels_wife, massa).
mother(ismaels_wife, mahalath).
mother(ismaels_wife, hadad).
mother(ismaels_wife, bashemath).
mother(bethuels_wife, laban).
mother(bethuels_wife, rebekah).
mother(lots_first_wife, first_daughter).
mother(lots_first_wife, second_daughter).
mother(first_daughter, moab).
mother(second_daughter, ben_ammi).
mother(bashemath, reuel).
mother(leah, levi3rd).
mother(leah, judas4th).
mother(mahalath, aliah).
mother(mahalath, elak).
mother(lebans_wife, rachel).
mother(lebans_wife, leah).
```

Family trees (just Datalog here) …

- husband(terah, terahs_first_wife).
  husband(terah, terahs_second_wife).
  husband(abraham, sarah).
  husband(abraham, hagar_concubine).
  husband(nahor, milcah).
  husband(haran, harans_wife).
  husband(isaac, rebekah).
  husband(ismael, ismaels_wife).
  husband(kemuel, kemuels_wife).
  husband(bethuel, bethuels_wife).
  husband(lot, lots_first_wife).
  husband(lot, first_daughter).
  husband(lot, second_daughter).
  husband(esau, bashemath).
  husband(jacob, leah).
  husband(jacob, rachel).
  husband(esau, mahalath).
  husband(laban, labans_wife).

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

convention in
these slides

Does husband(X,Y) mean
"X is the husband of Y"
or
"The husband of X is Y"?
Conventions vary … pick one and stick to it!

Family trees (just Datalog here) …

- % database
  mother(sarah,isaac).
  father(abraham,isaac).
  …
- parent(X, Y):- mother(X, Y).
  parent(X, Y):- father(X, Y).

- grandmother(X, Y):- mother(X, Z), parent(Z, Y).
  grandfather(X, Y):- father(X, Z), parent(Z, Y).
- grandparent(X, Y):- grandfather(X, Y).
  grandparent(X, Y):- grandmother(X, Y).

- Can we refactor this code on blackboard to avoid duplication?
  - better handling of male/female
    - currently grandmother and grandfather repeat the same “X…Z…Y” pattern
  - better handling of generations
    - currently great_grandmother and great_grandfather would repeat it again

Family trees (just Datalog here) …

- Refactored database (now specifies parent, not mother/father):
  - parent(sarah, isaac).   female(sarah).
  - parent(abraham, isaac).   male(abraham).

- Refactored ancestry (recursive, gender-neutral):
  - anc(0,X,X).
  - anc(N,X,Y) :- parent(X,Z), anc(N-1,Z,Y).

- Now just need one clause to define each English word:

  -
    ```
    parent(X,Y)        :- anc(1,X,Y).
    mother(X,Y)        :- parent(X,Y), female(X).
    father(X,Y)        :- parent(X,Y), male(X).
    ```

  -
    ```
    grandparent(X,Y)   :- anc(2,X,Y).
    grandmother(X,Y)   :- grandparent(X,Y), female(X).
    grandfather(X,Y)   :- grandparent(X,Y), male(X).
    ```

  -
    ```
    great_grandparent(X,Y)      :- anc(3,X,Y).
    etc.
    ```

Family trees (just Datalog here) …

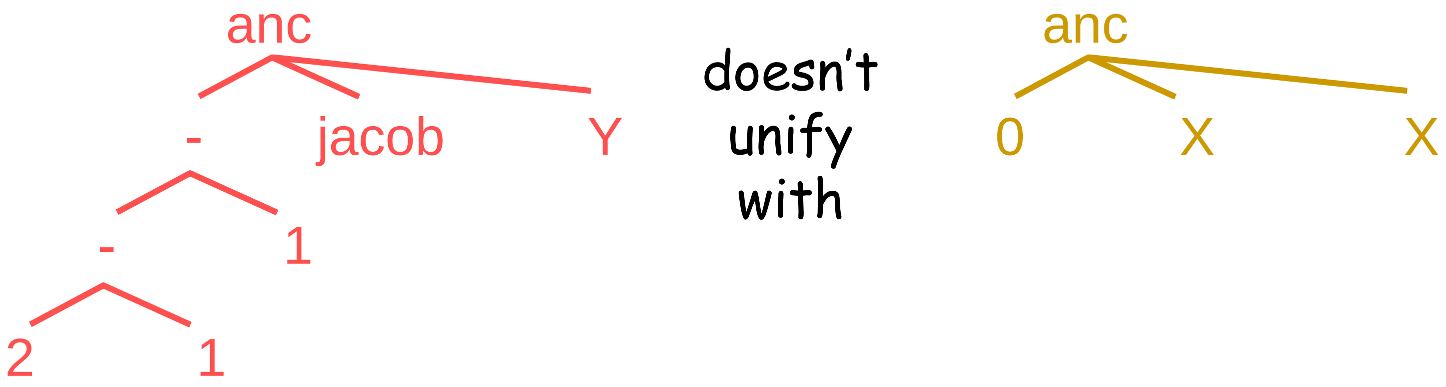
- Refactored ancestry (recursive, gender-neutral):
  - anc(0,X,X).
  - anc(N,X,Y) :- parent(X,Z), anc(N-1,Z,Y).

- Wait a minute! What does anc(2,abraham,Y) do?
  - Recurses on anc(2-1, isaac, Y).
  - Which recurses on anc((2-1)-1, jacob,Y).
  - Which recurses on anc(((2-1)-1)-1, joseph, Y). …

Family trees (just Datalog here) …

- Refactored ancestry (recursive, gender-neutral):
  - anc(0,X,X).
  - anc(N,X,Y) :- parent(X,Z), anc(N-1,Z,Y).

- Wait a minute! What does anc(2,abraham,Y) do?
  - Recurses on anc(2-1, isaac, Y).
  - Which recurses on anc((2-1)-1, jacob,Y).
    - Oops! (2-1)-1 isn’t zero. It’s ’-’(’-’(2,1),1)), a compound term.



Family trees (just Datalog here) …

- Refactored ancestry (recursive, gender-neutral):
  - anc(0,X,X).
  - anc(N,X,Y) :- ~~parent(X,Z), anc(N-1,Z,Y).~~

    N > 0, M is N-1, parent(X,Z), anc(M,Z,Y).

'is' does arithmetic for you:

'is'(0,1-1).	0 is 1-1.
'is'(4,2+2).	4 is 2+2.
'is'(24, 7*7-5*5)	24 is 7*7-5*5.

cuts off the search for grandchildren at 2 levels (once N <= 0, it's legal but wasteful to continue to recurse in hopes that we'll run into 0 again if we keep subtracting 1!)

Family trees (just Datalog here) …

- Refactored ancestry (recursive, gender-neutral):
  - anc(0,X,X).
  - anc(N,X,Y) :- M is N-1, parent(X,Z), anc(M,Z,Y).

- Now, the above works well for queries like

  anc(2,abraham,Y).   % query mode: anc(+,+,-)
  anc(2,X,jacob).   % query mode: anc(+,-,+)
  anc(2,X,Y).   % query mode: anc(+,-,-)

- But what happens if N is unassigned at query time?

  anc(N,abraham,jacob).   % query mode: anc(-,+,+)

  “Instantiation fault” on constraint “M is N-1.”

  The ’is’ built-in predicate doesn’t permit queries in the mode ’is’(-,-)!

  So can’t compute N-1.

  At least not without using an ECLiPSe delayed constraint: M #= N-1.

  A delayed constraint doesn’t have to be satisfied yet, but we’ll hang onto it for later. Anything we learn later about the domains of M and N will be propagated.

  Same problem if we have the constraint N > 0, which only allows ‘>’(+,+).

  Here the ECLiPSe delayed constraint would be N #> 0.

Family trees (just Datalog here) …

- Refactored ancestry (recursive, gender-neutral):
  - anc(0,X,X).
  - anc(N,X,Y) :- M is N-1, M >= 0, parent(X,Z), anc(M,Z,Y).

- Now, the above works well for queries like
  anc(2,abraham,Y). % query mode: anc(+,+,-)
  anc(2,X,jacob). % query mode: anc(+,-,+)
  anc(2,X,Y). % query mode: anc(+,-,-)
- But what happens if N is unassigned at query time?
  anc(N,abraham,jacob). % query mode: anc(-,+,+)
- For this case we wish we had written:
  - anc(0,X,X).
  - anc(N,X,Y) :- parent(X,Z), anc(M,Z,Y), N is M+1.
  - Here we query parent(+,-), which binds Z,
  - and then recursively query anc(-,+,+) again, which binds M,
  - and then query ’is’(-,+), which is a permitted mode for ‘is’. That works.
- What a shame that we have to write different programs to handle different query modes! Not very declarative.

A few more examples of family relations

(only the gender-neutral versions are shown)

- half_sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.
- sibling(X,Y) :- mother(Z,X), mother(Z,Y), father(W,X), father(W,Y), X \=Y.
  - Warning: This inequality constraint X \= Y only works right in mode +,+.
  - (It asks whether unification *would fail.* So the answer to A \= 4 is “no”, since A=4 would succeed! There is no way for Prolog to represent that A can be “anything but 4” – there is no “anything but 4” term. However, ECLiPSe can use domains or delayed constraints to represent this property of A: use a delayed constraint A #\= 4.)
- aunt_or_uncle(X,Y) :- sibling(X,Z), parent(Z,Y).
- cousin(X,Y):- parent(Z,X), sibling(Z,W), parent(W,Y).
- deepcousin(X,Y):- sibling(X,Y). % siblings are 0th cousins
- deepcousin(X,Y):- parent(Z,X), deepcousin(Z,W), parent(W,Y).

  % we are Nth cousins if we have parents who are (N-1)st cousins

Ancestry

- deepcousin(X,Y):- sibling(X,Y). % siblings are 0th cousins
- deepcousin(X,Y):- parent(Z,X), deepcousin(Z,W), parent(W,Y).

  % we are Nth cousins if we have parents who are (N-1)st cousins

- Suppose we want to count the cousin levels.
- nth_cousin(N,X,Y) :- …?
  - Should remind you of a previous problem: work it out!
  - What is the base case?
  - Who are my 3rd cousins? query mode +,+,-
  - For what N are we Nth cousins? query mode -,+,+
- Did you ever wonder what “3rd cousin twice removed” means?
  - answer(X,Y) :- nth_cousin(3,X,Z), anc(2,Z,Y).

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]
- $\text{if } X=\underbrace{[3,4]}_{\text{cons(3,cons(4,nil))}}\text{, then } \underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}}=\overbrace{[1,2,3,4]}^{\text{cons(1, cons(2, cons(3, cons(4, nil))))}}$

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]

- $\overbrace{[1,2,3,4]}^{\text{cons(1, cons(2, cons(3, cons(4, nil))))}}=\underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}} \rightarrow \underbrace{X=[3,4]}_{\text{cons(3,cons(4,nil))}}$ by unification

Lists

- How do you represent the list 1,2,3,4?
- Use a structured term:
  cons(1, cons(2, cons(3, cons(4, nil))))
- Prolog lets you write this more prettily as [1,2,3,4]

- $\overbrace{[1,2]}^{\text{cons(1, cons(2, nil))}} \quad = \underbrace{[1,2|X]}_{\text{cons(1,cons(2,X))}} \rightarrow \quad \underbrace{X=[]}_{\text{nil}}$

Decomposing lists

- first(X,List) :- ...?

- first(X,List) :- List=[X|Xs].
  - Traditional variable name: "X followed by some more X's."
- first(X, [X|Xs]).
  - Nicer: eliminates the single-use variable List.
- first(X, [X|_]).
  - Also eliminate the single-use variable Xs.

Decomposing lists

- first(X,   [X|_]).
- rest(Xs, [_|Xs]).

- Query: first(8, [7,8,9]).
  - Answer: no
- Query: first(X, [7,8,9]).
  - Answer: X=7
- Query: first(7, List).
  - Answer: List=[7|Xs]
    (will probably print an internal var name like _G123 instead of Xs)
- Query: first(7, List), rest([8,9], List).
  - Answer: List=[7,8,9].
  - *Can you draw the structures that get unified to do this?*

Decomposing lists

- In practice, no one ever actually defines rules for “first” and “rest.”
- Just do the same thing by pattern matching: write things like [X|Xs] directly in your other rules.

List processing: **member**

- member(X,Y) should be true if X is any object, Y is a list, and X is a member of the list Y.

- member(X, [X|_]). % same as "first"
- member(X, [Y|Ys]) :- member(X,Ys).
- Query: member(giraffe, [beaver, ant, steak(giraffe), fish]).
  - Answer: no (why?)
  - It's recursive, but where is the base case???

    -
      ```
      if (list.empty()) then return "no"               % missing in Prolog??
      else if (x==list.first()) then return "yes"      % like 1st Prolog rule
      else return member(x, list.rest())               % like 2nd Prolog rule
      ```

question thanks to Michael J. Ciaraldi and David Finkel

List processing: **member**

- Query: member(X, [7,8,7]).
  - Answer: X=7 ;
    X=8 ;
    X=7
- Query: member(7, List).
  - Answer: List=[7 | Xs] ;
    List=[X1, 7| Xs] ;
    List=[X1, X2, 7 | Xs] ;
    …   (willing to backtrack forever)

List processing: **length**

- Query: member(7, List), member(8,List), length(List, 3).
  - Answer: List=[7,8,X] ;
    List=[7,X,8] ;
    (now searches forever for next answer
    – see prev. slide!)
- Query: length(List, 3), member(7, List), member(8,List).
  - Answer: List=[7, 8, X] ;
    List=[7, X, 8] ;
    List=[8, 7, X] ;
    List=[X, 7, 8] ;
    List=[8, X, 7] ;
    List=[X, 8, 7]

    (why in this order?)

- How do we define length?
- length([], 0).
- length([_|Xs],N) :-
  length(Xs,M), N is M+1.
- But this will cause infinite recursion for length(List,3).

List processing: **length**

- Query: member(7, List), member(8,List), length(List, 3).
  - Answer: doesn’t terminate (see previous slide!)
- er(7, List), member(8,List).

- How do we define length?
- length([], 0).
- length([_|Xs],N) :- N > 0,
  length(Xs,M), N is M+1.
- But this will cause an instantiation fault when we recurse. We’ll try to test M > 0, but M is still unbound.

~~How do we define length?~~

~~length([], 0).~~

~~length([_|Xs],N) :-~~
~~length(Xs,M), N is M+1.~~

~~But this will cause infinite recursion for length(List,3).~~

List

- Que List, 3).
  - Ar
- H ist).

- How do we define length?
- length([], 0).
- length([_|Xs],N) :- N > 0,
  M is N-1, length(Xs,M).
- Works great for length(List,3).
- Unfortunately, instantiation fault for length([a,b,c],N).
  For that case we should use our first version!

- Prolog does have hacky ways to tell which case we're in. So we can have <u>both</u> definitions … built-in version of "length" does.

- ~~H~~
- ~~length([], 0).~~
- ~~length([_|Xs],N) :- N > 0,~~
  ~~length(Xs,M), N is M+1.~~
- ~~But this will cause an instantiation fault when we recurse. We'll try to test M > 0, but M is still unbound.~~

~~How do we define length?~~

~~length([], 0).~~

~~length([_|Xs],N) :-~~
~~length(Xs,M), N is M+1.~~

~~But this will cause infinite recursion for length(List,3).~~

List

- How do we define length?
- length([], 0).
- length([_|Xs],N) :- N > 0,
  M is N-1, length(Xs,M).
- Works great for length(List,3).
- Unfortunately, instantiation fault for length([a,b,c],N).
  For that case we should use our first version!

- Prolog does have hacky ways to tell which case we're in. So we can have both definitions … built-in version of "length" does.

- Que
  - Ar

List, 3).

- H
  - length([], 0)

ist).

Toto, I don't think we're in declarative programming anymore …

The problem:

N is M+1 is not "pure Prolog."

Neither is N > 0.

- These constraints can't be processed by unification as you encounter them. They're handled by some outside mechanism that requires certain variables to be already assigned.
- Is there a "pure Prolog" alternative (maybe slower, but always works)?

Arithmetic in pure Prolog

- Let’s rethink arithmetic as term unification!
- I promised we’d divide 6 by 2
  by making Prolog prove that ∃x 2*x = 6.
- Query: times(2,X,6). So how do we program times?

Represent 0 by z (for “zero”)

Represent 1 by s(z) (for “successor”).

Represent 2 by s(s(z))

Represent 3 by s(s(s(z)))

… “Peano integers”

- So actually our query times(2,X,6) will be written
  times(s(s(z)), X, s(s(s(s(s(s(z))))))).

A pure Prolog definition of **length**

- length([ ],z).
- length([_|Xs], s(N)) :- length(Xs,N).

- This is pure Prolog and will work perfectly everywhere.
- Yeah, it’s a bit annoying to use Peano integers for input/output:
  - Query: length([[a,b],[c,d],[e,f]], N).
    Answer: N=s(s(s(z)))
  - Query: length(List, s(s(s(z)))).
    Answer: List=[A,B,C]

yuck?

- But you could use impure Prolog to convert them to “ordinary” numbers just at input and output time …

A pure Prolog definition of **length**

- length([ ],z).
- length([_|Xs], s(N)) :- length(Xs,N).

- This is pure Prolog and will work perfectly everywhere.
- Converting between Peano integers and ordinary numbers:
  - Query: length([[a,b],[c,d],[e,f]], N), decode(N,D).
    Answer: N=s(s(s(z))), D=3
  - Query: encode(3,N), length(List, N).
    Answer: N=s(s(s(z))), List=[A,B,C]

- decode(z,0).   decode(s(N),D) :- decode(N,E), D is E+1.
- encode(0,z).   encode(D,s(N)) :- D > 0, E is D-1, encode(E,N).

2+2 in pure Prolog

- First, let's define a predicate add/3.
- add(z,B,B). % 0+B=B.
- add(s(A),B,Sum) :- add(A,s(B),Sum). % (A+1)+B=S
  ← A+(B+1)=S.
- The above should make sense declaratively.
- Don't worry yet about how the solver works.
- Just worry about what the program says.
- It inductively defines addition of natural numbers! The first line is the base case.

2+2 in pure Prolog

- First, let’s define a predicate add/3.
- add(z,B,B). % 0+B=B.
- add(s(A),B,Sum) :- add(A,s(B),Sum). % (A+1)+B=S ← A+(B+1)=S.

add(s(s(z)),s(s(z)),Sum) *original query*

2+2 in pure Prolog

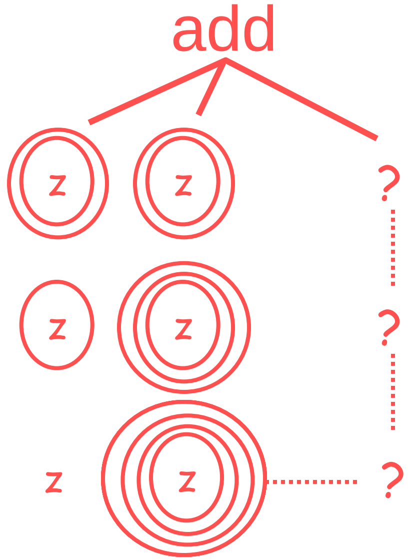
- First, let’s define a predicate add/3.
- add(z,B,B). % 0+B=B.
- add(s(A),B,Sum) :- add(A,s(B),Sum). % (A+1)+B=S
  ← A+(B+1)=S.

add(z , z , ?) original query

2+2 in pure Prolog

- First, let’s define a predicate add/3.
- add(z,B,B). % 0+B=B.
- add(s(A),B,Sum) :- add(A,s(B),Sum). % (A+1)+B=S

  ← A+(B+1)=S.



note the unification of variables between different calls

original query matches head of rule

1st recursive call matches head of rule

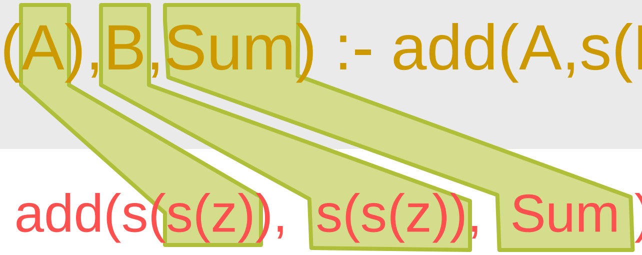
2nd recursive call matches base case

Removed outer skins from 1st argument (outside-in), wrapping them around 2nd argument (inside-out).

2+2 in pure Prolog

- First, let’s define a predicate add/3.
- add(z,B,B). % 0+B=B.
- add(s(A),B,Sum) :- add(A,s(B),Sum). % (A+1)+B=S

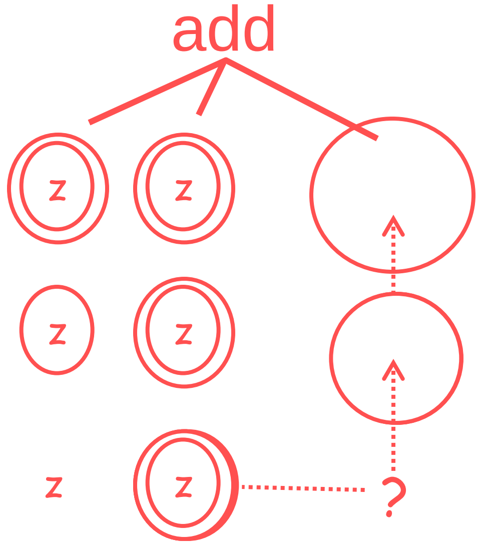
  ← A+(B+1)=S.



- Query: add(s(s(z)), s(s(z)), Sum). % 2+2=?
  - Matches head of second clause: A=s(z), B=s(s(z)).
  - So now we have to satisfy body: add(s(z), s(s(s(z))), Sum).
    - Matches head of second clause: A=z, B=s(s(s(z))).
    - So now we have to satisfy body: add(z, s(s(s(s(z)))), Sum).
      - Matches head of first clause: B=s(s(s(s(z)))), B=Sum.
      - So Sum=s(s(s(s(z))))! Unification has given us our answer.

More 2+2: An interesting variant

- First, let’s define a predicate add/3.
- add(z,B,B). % 0+B=B.
- add(s(A),B,s(Sum)) :- add(A,B,Sum). % (A+1)+B=(S+1) ← A+B=S.



original query matches head of rule

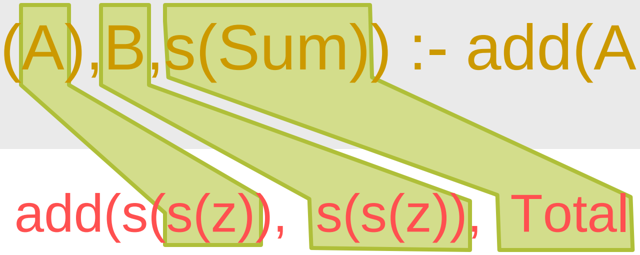
1st recursive call matches head of rule

2nd recursive call matches base case

Removed outer skins from 1st argument (outside-in), nested them to form the result (outside-in), dropped 2nd argument into the core.

More 2+2: An interesting variant

- First, let’s define a predicate add/3.
- add(z,B,B). % 0+B=B.



- add(s(A),B,s(Sum)) :- add(A,B,Sum). % (A+1)+B=(S+1)
  ← A+B=S.
- Query: add(s(s(z)), s(s(z)), Total). % 2+2=?
  - Matches head of second clause: A=s(z), B=s(s(z)), Total=s(Sum).
  - So now we have to satisfy body: add(s(z), s(s(z)), Sum).
    - Matches head of 2nd clause: A=z, B=s(s(z)), Total=s(s(Sum)).
    - So now we have to satisfy body: add(z, s(s(z)))), Sum).
      - Matches head of <u>first</u> clause: B=s(s(z)).
      - So we have built up Total=s(s(Sum))=s(s(s(z))).

An amusing query

- Query: add(z, N, s(N)).   % 0+N = 1+N
  - Answer: you would expect “no”
  - But actually: N = s(s(s(s(s(s(…))))))
    - Looks good: 0+∞ = 1+∞ since both are ∞ !

    - Only get this circular term since Prolog skips the occurs check while unifying the query with add(z,B,B)

List processing continued: **append**

- You probably already know how to write a non-destructive append(Xs,Ys) function in a conventional language, using recursion.

- append(Xs,Ys):

```
append(Xs,Ys):
if (Xs.empty())
    return Ys
else
    subproblem = Xs.rest();   // all but the 1st element
    subsolution = append(subproblem, Ys)
    return cons(Xs.first(), subsolution)
```

List processing continued: **append**

- You probably already know how to write a non-destructive append(Xs,Ys) function in a conventional language, using recursion.
- In more Prologgy notation:
- append([],Ys): return Ys
- append([X|Xs],Ys): return [X | append(Xs,Ys)]

List processing continued: **append**

- You probably already know how to write a non-destructive append(Xs,Ys) function in a conventional language, using recursion.

- In actual Prolog, the function looks much the same, but once you've written it, you can also run it backwards!
- In Prolog there are no return values. Rather, the return value is a third argument: append(Xs,Ys,Result).
- This is a constraint saying that Result must be the append of the other lists.
- Any of the three arguments may be known (or partly known) at runtime. We look for satisfying assignments to the others.

List processing continued: **append**

- append(Xs,Ys,Result) should be true if Xs and Ys are lists and Result is their concatenation (another list).
- Query: append([1,2],[3,4],Result)
  - Answer: Result=[1,2,3,4]
- Try this:
  - append([],Ys,Ys).
  - append([X|Xs],Ys,Result) :- … ?

List processing continued: **append**

- append(Xs,Ys,Result) should be true if Xs and Ys are lists and Result is their concatenation (another list).
- Query: append([1,2],[3,4],Result)
  - Answer: Result=[1,2,3,4]
- Try this:
  - append([],Ys,Ys).
  - append([X|Xs],Ys,Result) :- append(Xs,[X|Ys],Result).
  - But wait: what order are the onion skins being wrapped in?
  - This is like the first version of 2+2 …

List processing continued: **append**

- append(Xs,Ys,Result) should be true if Xs and Ys are lists and Result is their concatenation (another list).
- Query: appendrev([1,2],[3,4],Result)
  - Answer: Result=[2,1,3,4]
- Rename this to appendrev!
  - appendrev([],Ys,Ys).
  - appendrev([X|Xs],Ys,Result) :- appendrev(Xs,[X|Ys],Result).
  - *But wait: what order are the onion skins being wrapped in?*
  - *This is like the first version of 2+2 …*

List processing continued: **append**

- Let's wrap the onion skins like the other 2+2 …

- Query: append([1,2],[3,4],Result)
  - Answer: Result=[1,2,3,4]
- Here's the correct version of append:
  - append([],Ys,Ys).
  - $$\text{append}(\underbrace{[X \mid Xs],Ys,}_{\text{1. our inputs}}\underbrace{[X \mid Result]}_{\text{4. construct our output}})\text{ :- append}(\underbrace{Xs,Ys,}_{\text{2. inputs to recursive call}}\underbrace{Result}_{\text{3. output of recursive call}}).$$

A procedural (non-declarative) way to read this rule

List processing continued: **append**

- Let's wrap the onion skins like the other 2+2 …

- Query: append([1,2],[3,4],Result)
  - Answer: Result=[1,2,3,4]
- Here's the correct version of append:
  - append([],Ys,Ys).
  - append([X|Xs],Ys,[X|Result]) :- append(Xs,Ys,Result).
  - This version also makes perfect sense declaratively.
  - And we still have a use for the other version, append<u>rev</u>:
    - reverse(Xs,Ys) :- appendrev(Xs,[],Ys).

Arithmetic continued: Subtraction

```
add(z,B,B).                                  % 0+B=B.
add(s(A),B,Sum) :- add(A,s(B),Sum).          % (A+1)+B=S
                                                   ← A+(B+1)=S.
```

```
add(z,B,B).                                  % 0+B=B.
add(s(A),B,s(Sum)) :- add(A,B,Sum).          % (A+1)+B=(S+1)
                                                   ← A+B=S.
```

- add(s(s(z)), X, s(s(s(s(s(z)))))).
- add(s(s(s(s(s(z))))), X, s(s(z))).

- Pure Prolog gives you subtraction for free!

Multiplication and division

- How do you define multiplication?
- (Then division will come for free.)

Square roots

- mult(X, X, s(s(s(s(s(s(s(s(s(z)))))))))).

More list processing: Sorting

- sort(Xs, Ys)

- You can write recursive selection sort, insertion sort, merge sort, quick sort … where the list Xs is completely known so that you can compare its elements using <.
- This is basically like writing these procedures in any functional language (LISP, OCaml, …). It's no more declarative than those languages.
- But how about this more declarative version?
  - sort(Xs, Ys) :- permutation(Xs,Ys), ordered(Ys).
- How do we write these?
  - ordered is the easy one …

More list processing: Sorting

- ordered([]).
- ordered([X]).
- ordered([X,Y|Ys]) :- … ?

More list processing: Sorting

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

More list processing: Sorting

- Query: deleteone(b, [a,b,c,b], Xs).
- Answer: Xs=[a,c,b] ;
  Xs=[a,b,c]

- deleteone(X,[X|Xs],Xs).
- deleteone(Z,[X|Xs],[X|Ys]) :-
      deleteone(Z,Xs,Ys).

More list processing: Sorting

- Can we use deleteone(X,List,Rest) to write permutation(Xs,Ys)?
- permutation([], []).
- permutation(Xs, [Y|PYs]) :-
  ```
        deleteone(Y,Xs,Ys),
        permutation(Ys,PYs).
  ```
- "Starting with Xs, delete any Y to leave Ys. Permute the Ys to get PYs. Then glue Y back on the front."
- To repeat, sorting by checking all permutations is horribly inefficient. You can also write the usual fast sorting algorithms in Prolog.
  - Hmm, but we don't have random-access arrays … and it's hard to graft those on if you want the ability to modify them …
  - Can use lists rather than arrays if your algorithm is selection sort, insertion sort, mergesort … try these yourself in Prolog!

Mergesort

- Query: mergesort([4,3,6,5,9,1,7],S).
- Answer: S=[1,3,4,5,6,7,9]

```
mergesort([],[]).
mergesort([A],[A]).
mergesort([A,B|R],S) :-
        split([A,B|R],L1,L2),
        mergesort(L1,S1),  mergesort(L2,S2),
        merge(S1,S2,S).
```

```
split([],[],[]).
split([A],[A],[]).
split([A,B|R],[A|Ra],[B|Rb]) :-  split(R,Ra,Rb).
```

```
merge(A,[],A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :-  A =< B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :-  A > B,  merge([A|Ra],Rb,M).
```

A bad SAT solver

(no short-circuit evaluation or propagation)

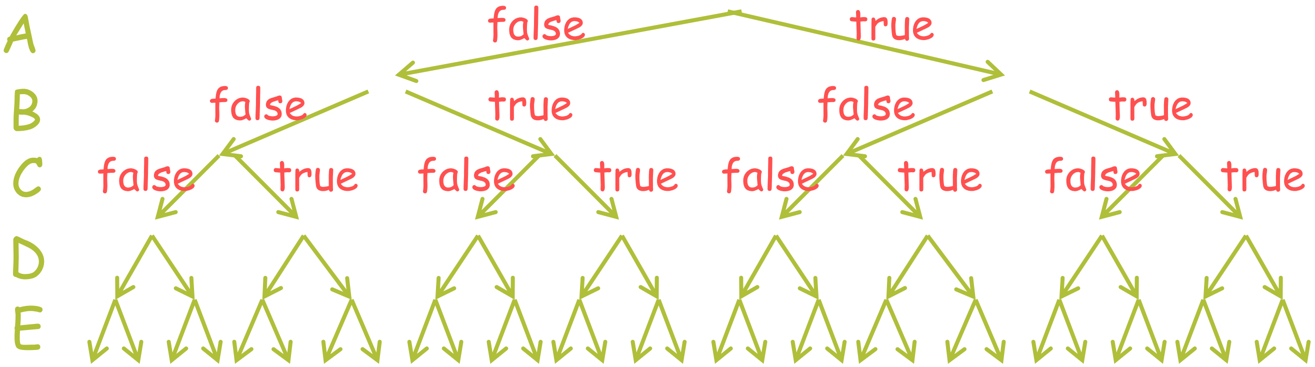
```
// Suppose formula uses 5 variables: A, B, C, D, E
for $A \in \{0, 1\}$
    for $B \in \{0, 1\}$
        for $C \in \{0, 1\}$
            for $D \in \{0, 1\}$
                for $E \in \{0, 1\}$
                    if formula is true
                        immediately return (A,B,C,D,E)
return UNSAT
```

A bad SAT solver in Prolog

- Query (what variable & value ordering are used here?)
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - % formula (A v ~C v D) ^ (~B v C v E) ^ (A xor E) ^ …
  - formula(A,B,C,D,E) :-
    clause1(A,C,D), clause2(B,C,E), xor(A,E), …
  - % clauses in that formula
  - clause1(true,_,_). clause1(_,false,_). clause1(_,_,true).
  - clause2(false,_,). clause2(_,true,_). clause2(_,_,true).
  - xor(true,false). xor(false,true).

A bad SAT solver in Prolog

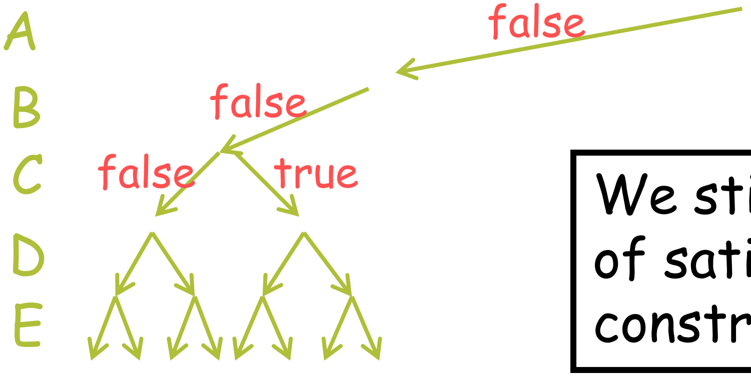
- Query (what variable & value ordering are used here?)
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).



The Prolog cut operator, “!”

- Query
  - bool(A),bool(B), !, bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for b
  - bool(false). bool(true).
  - …

Cuts off part of the search space.
Once we have managed to satisfy
bool(A),bool(B) and gotten past !,
we are committed to our choices so far
and won’t backtrack to revisit them.

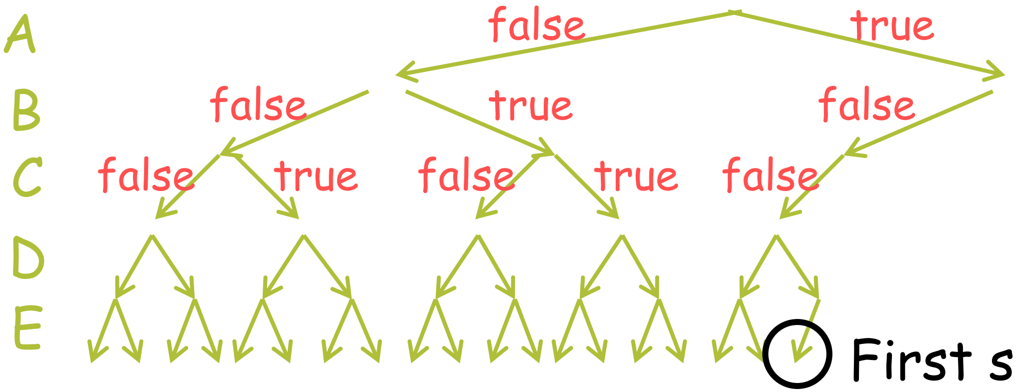


We still backtrack to find other ways
of satisfying the subsequent
constraints bool(C),bool(D),…

The Prolog cut operator, “!”

- Query
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E), **!**.
- Program
  - % values available for b
  - bool(false). bool(true).
  - …

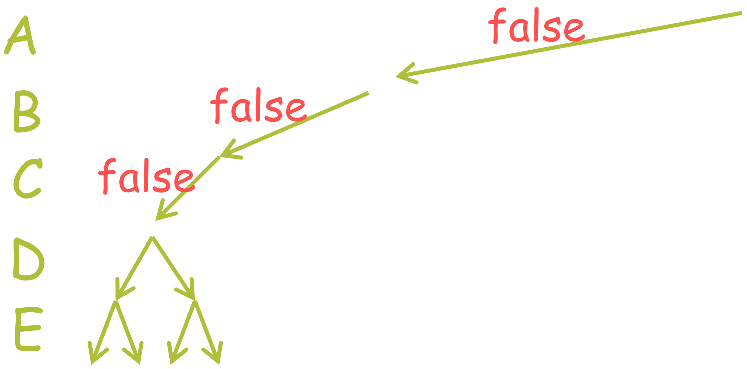
Cuts off part of the search space. Once we have managed to satisfy the constraints before ! (all constraints in this case), we don't backtrack. So we return only first satisfying assignment.



First satisfying assignment

The Prolog cut operator, “!”

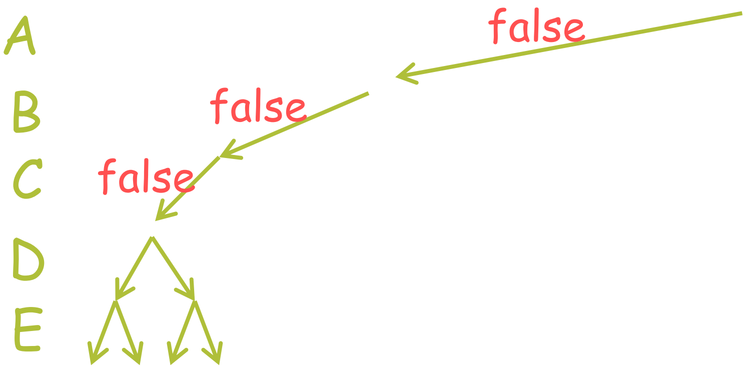
- Query
  - bool(A),bool(B),bool(C),!,bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - …



The Prolog cut operator, “!”

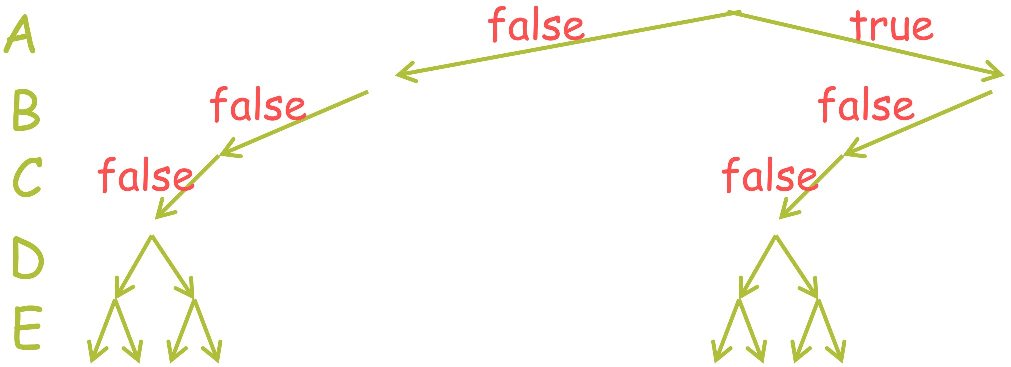
- Query
  - bool(A), bool2(B,C), !,bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - bool2(X,Y) :- bool(X), bool(Y).

Same effect, using a subroutine.



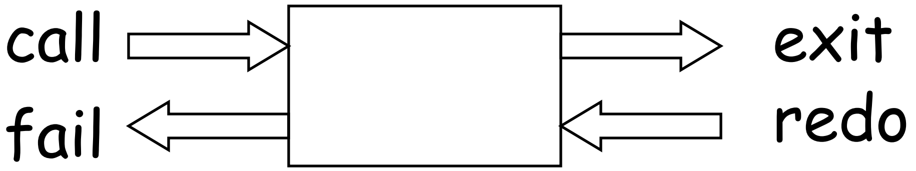
The Prolog cut operator, "!"

- Query
  - bool(A), bool2(B,C), ,bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking searc
  - bool(false). bool(true).
  - bool2(X,Y) :- bool(X), bool(Y), !.
    - % equivalent to: bool2(false,false).



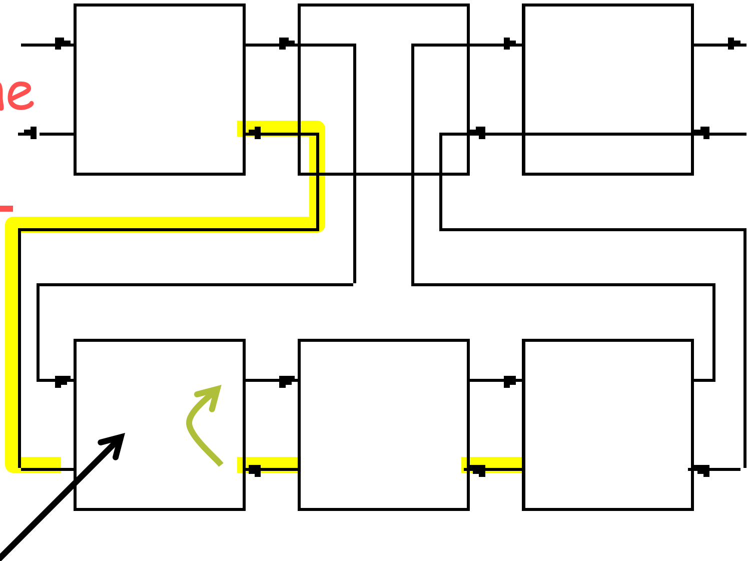
Now effect of "!"
is local to bool2.
bool2 will commit to
its first solution,
namely (false,false),
not backtracking to
get other solutions.
But that's just how
bool2 works inside.
Red query doesn't
know bool2 contains a
cut; it backtracks to
try different A,
calling bool2 for each.

How cuts affect backtracking



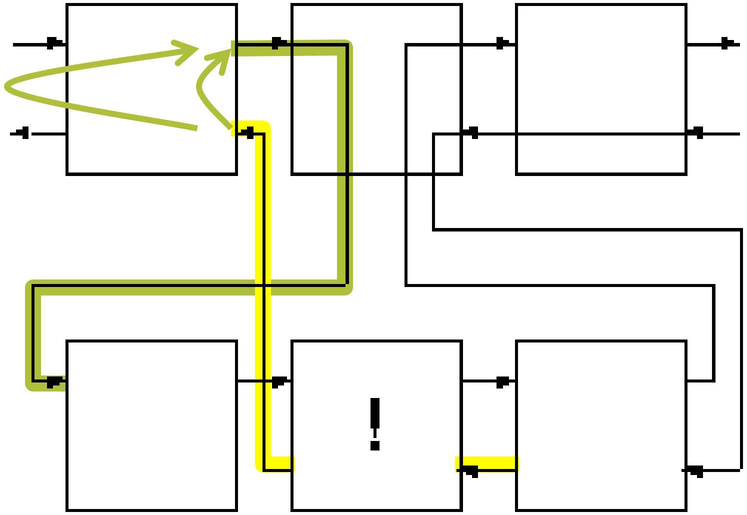
main routine

subroutine for clause #2



Can try other options here before failing and returning to caller

Normal backtracking if we fail within clause #2



But fail immediately (return to caller) if we backtrack past a cut.

Caller can still go back & change something & call us again.

A bad SAT solver in Prolog

- Query
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - % formula (A v ~C v D) ^ (~B v C v E) ^ (A xor E) ^ …
  - formula(A,B,C,D,E) :-
    clause1(A,C,D), clause2(B,C,E), xor(A,E), …
  - clause1(true,_,_).
  - clause1(_,false,_).
  - clause1(_,_,true).

Truly inefficient!
Even checking whether the formula is satisfied may take exponential time, because we backtrack through all the ways to justify that it's satisfied!

A bad SAT solver in Prolog

- Query
  - bool(A),bool(B),bool(C),bool(D),bool(E),formula(A,B,C,D,E).
- Program
  - % values available for backtracking search
  - bool(false). bool(true).
  - % formula (A v ~C v D) ^ (~B v C v E) ^ (A xor E) ^ …
  - formula(A,B,C,D,E) :-
    clause1(A,C,D), clause2(B,C,E), xor(A,E), …
  - clause1(true,_,_) :- !.
  - clause1(_,false,_) :- !. ←
  - clause1(_,_,true).

Much better. Now once we know that clause1 is satisfied, we can move on; we don't have to backtrack through all the reasons it's satisfied.

Are these "green cuts" that don't change the output of the program?
Yes, in this case, if we only call clause1 in mode clause1(+,+,+).
Except that they will eliminate duplicate solutions, too.

Another pedagogical example of cut

```
eats(sam, dal).                 eats(josie, samosas).
eats(sam, curry).               eats(josie, curry).
eats(rajiv, burgers).           eats(rajiv, dal).
compatible(Person1, Person2) :- eats(Person1, Food),
                                eats(Person2, Food).
compatible(Person1, Person2) :- watches(Person1, Movie),
                                watches(Person2, Movie).
```

- To whom should we advertise curry?
- eats(X,curry), compatible(X,Y).
  - X=sam, Y=sam; X=sam, Y=josie; X=josie, X=sam; X=josie, Y=josie
- eats(X,curry), !, compatible(X,Y).
  - X=sam, Y=sam; X=sam, Y=josie
- eats(X,curry), compatible(X,Y), !.
  - X=sam, Y=sam

Using cut to force determinism

- Query: deleteone(b, [a,b,c,b], Xs).
- Answer: Xs=[a,c,b] ;
  Xs=[a,b,c]
- deleteone(X,[X|Xs],Xs).
- deleteone(Z,[X|Xs],[X|Ys]) :- deleteone(Z,Xs,Ys).

Using cut to force determinism

deletefirst

- Query: ~~deleteone~~(b, [a,b,c,b], Xs).
- Answer: Xs=[a,c,b] ;
  ~~Xs=[a,b,c]~~
- deletefirst(X,[X|Xs],Xs) :- ! .
- deletefirst(Z,[X|Xs],[X|Ys]) :- deletefirst(Z,Xs,Ys).

Using cut to override default rules with specific cases

- permissions(superuser, File, [read,write]) :- !.
- permissions(guest, File, [read]) :- public(File), !. % exception to exception
- permissions(guest, File, []) :- !. % if this matches, prevent lookup
- permissions(User, File, PermissionsList) :- lookup(…).

  % unsafe? what if looked-up permissions were set wrong?

- can_fly(X) :- penguin(X), !, fail.
- can_fly(X) :- bird(X).

- progenitor(god, adam) :- !.   % cut is unnecessary but efficient
- progenitor(god, eve) :- !.   % cut is unnecessary but efficient.
- progenitor(X,Y) :- parent(X,Y).

Using cut to get negation, sort of

```
eats(sam, dal).                   eats(josie, samosas).
eats(sam, curry).                 eats(josie, curry).
eats(rajiv, burgers).             eats(rajiv, dal).
```

- \+ eats(sam,dal).   % \+ means “not provable”
  - No
- \+ eats(sam,rutabaga).
  - Yes
- \+ eats(sam,X).
  - No   % since we can prove that sam does eat some X
- \+ eats(robot,X).
  - Yes   % since we can’t currently prove that robot eats anything

Using cut to get negation, sort of

```
eats(sam, dal).                    eats(josie, samosas).
eats(sam, curry).                  eats(josie, curry).
eats(rajiv, burgers).              eats(rajiv, dal).
avoids(Person,Food) :- eats(Person,Food), !, fail.
avoids(Person,Food).
```

- avoids(sam,dal).   % “avoids” is implemented in the same way as \+
  - No
- avoids(sam,rutabaga).
  - Yes
- avoids(sam,X).
  - No   % since we can prove that sam does eat some X
- avoids(robot,X).
  - Yes   % since we can’t currently prove that robot eats anything

If we can prove “eats,” we commit with !
to not being able to prove “avoid”
Otherwise we can prove “avoid”!

More list processing: **deleteall**

- Query: deleteall(2, [1,2,3,1,2], Ys).
- Answer: Ys=[1,3,1]

- deleteall(X,[X|Xs],Ys) :- deleteall(X,Xs,Ys).
- deleteall(Z,[X|Xs],[X|Ys]) :- Z\=X, deleteall(Z,Xs,Ys).
- deleteall(Z,[],[]).

Works fine for ground terms :
2 \= 1, so we don't delete 1.

- But how about deleteall(Z, [1,2,3,1,2], Ys)?
- We'd like \= to mean "constrained not to unify."
  - So Z \= 1 should mean "Z can be any term at all except for 1."
  - But how do we represent that in memory??
  - Not like unification, which just specializes a variable to refer to a more specific term than before. "Anything but 1" is not a term.
- So instead, it means "these don't unify right now"
  - "Z \= 1" is just short for "\+ (Z=1)"

More list processing: **deleteall**

- Query: deleteall(A, [1,2,3,1,2], Ys).
- Answer: A=1, Ys=[2,3,2] since only first clause succeeds (and then A is ground in recursive call)
- deleteall(X,[X|Xs],Ys) :- deleteall(X,Xs,Ys).
- deleteall(Z,[X|Xs],[X|Ys]) :- Z\=X, deleteall(Z,Xs,Ys).
- deleteall(Z,[],[]).

- We'd like \= to mean "constrained not to unify."
  - So Z \= 1 should mean "Z can be any term at all except for 1."
  - But how do we represent that in memory??
  - Not like unification, which just specializes a variable to refer to a more specific <u>term</u> than before. "Anything but 1" is not a <u>term</u>.
- So instead, it means "these don't unify right now"
  - "Z \= 1" is just short for "\+ (Z=1)"

More list processing: **deleteall**

- Query: deleteall(A, [1,2,3,1,2], Ys).
- Answer: A=1, Ys=[2,3,2]

Equivalent way to make only 1st clause succeed (but faster: never tries 2nd)

- deleteall(X,[X|Xs],Ys) :- !, deleteall(X,Xs,Ys).
- deleteall(Z,[X|Xs],[X|Ys]) :- deleteall(Z,Xs,Ys).
- deleteall(Z,[],[]).

- We'd like \= to mean "constrained not to unify."
  - So Z \= 1 should mean "Z can be any term at all except for 1."
  - But how do we represent that in memory??
  - Not like unification, which just specializes a variable to refer to a more specific term than before. "Anything but 1" is not a term.
- So instead, it means "these don't unify right now"
  - "Z \= 1" is just short for "\+ (Z=1)"

More list processing: **deleteall**

- Query: deleteall(A, [1,2,3,1,2], Ys).
- Answer: A=1, Ys=[2,3,2] ;
  Instantiation fault   since =\= only allowed in mode +,+
- deleteall(X,[X|Xs],Ys) :- deleteall(X,Xs,Ys).
- deleteall(Z,[X|Xs],[X|Ys]) :- Z=\=X, deleteall(Z,Xs,Ys).
- deleteall(Z,[],[]).

- We'd like \= to mean "constrained not to unify."
  - So Z \= 1 should mean "Z can be any term at all except for 1."
  - But how do we represent that in memory??
  - Not like unification, which just specializes a variable to refer to a more specific term than before. "Anything but 1" is not a term.
- So instead, it means "these don't unify right now"
  - "Z \= 1" is just short for "\+ (Z=1)"

More list processing: **deleteall**

- Query: member(A,[1,2,3,1,2]), deleteall(A, [1,2,3,1,2], Ys).
- Answer: A=1, Ys=[2,3,2] ;
  A=2, Ys=[1,3,1] ; etc.

Ensure that A is ground
before we try calling deleteall
(5 answers)

- deleteall(X,[X|Xs],Ys) :- deleteall(X,Xs,Ys).
- deleteall(Z,[X|Xs],[X|Ys]) :- Z\=X, deleteall(Z,Xs,Ys).
- deleteall(Z,[],[]).

- We'd like \= to mean "constrained not to unify."
  - So Z \= 1 should mean "Z can be any term at all except for 1."
  - But how do we represent that in memory??
  - Not like unification, which just specializes a variable to refer to a more specific term than before. "Anything but 1" is not a term.
- So instead, it means "these don't unify right now"
  - "Z \= 1" is just short for "\+ (Z=1)"

More list processing: **deleteall**

- Query: deleteall(A, [1,2,3,1,2], Ys).
- Answer: A=1, Ys=[2,3,2] ;
  A=2, Ys=[1,3,1] ; etc.

ECLiPSe delayed constraint!
Will be handled once Z is known.

- deleteall(X,[X|Xs],Ys) :- deleteall(X,Xs,Ys).
- deleteall(Z,[X|Xs],[X|Ys]) :- Z#\=X, deleteall(Z,Xs,Ys).
- deleteall(Z,[],[]).

This is the "right" approach (fully declarative). Beyond Prolog. :-lib(ic).
How many answers? Still 5 answers?
Nope! 4 answers:

A=1, Ys=[2,3,2] ; match 1st clause (so A=1)
A=2, Ys=[1,3,1] ; match 2nd clause (so A#\=1), then 1st (so A=2)
A=3, Ys=[1,2,1,2] ; match 2nd (so A#\=1), then 2nd (A#\=2), then 1st (A=3)

If we match 2nd clause once more, then we'll have to keep matching it for the rest of the list, since we will have constraints $A \notin \{1,2,3\}$ that prevent us from taking the 1st clause again

A=A, Ys=[1,2,3,1,2], plus delayed goals saying $A \notin \{1,2,3\}$
match 2nd clause 5 times

More list processing: **deleteall**

- Query: deleteall(A, [1,2,3,1,2], Ys).
- Answer: A=1, Ys=[2,3,2] ;
  A=2, Ys=[1,3,1] ; etc.
- deleteall(X,[X|Xs],Ys) :- deleteall(X,Xs,Ys).
- deleteall(Z,[X|Xs],[X|Ys]) :- Z#\=X, deleteall(Z,Xs,Ys).
- deleteall(Z,[],[]).

ECLiPSe delayed constraint!
Will be handled once Z is known.

This is the "right" approach (fully declarative). Beyond Prolog. :-lib(ic).

Well, still not perfect. What happens with query
deleteall(1, List, [2,3,2])?
Unfortunately we get infinite recursion on the first clause.

Constraint logic programming …

- In constraint logic programming, you can include constraints on integers like N #= M+1 (rather than N is M+1) and X#\=Z (rather than X \=Z) without having to worry about which variables are already instantiated.

- If a constraint can't be processed yet, it will be handled later, as soon as its variables are sufficiently instantiated. Example: N #= M+1, N #= 5.
- In fact, do bounds propagation. Example: N #= M+1, N #> 5.

- But what happens if vars are *never* sufficiently instantiated?
- The **labeling(Vars)** constraint does backtracking search:
  - tries all assignments of Vars consistent with constraints so far
  - finds these assignments using backtracking search interleaved with constraint propagation (e.g., bounds consistency)
  - you can control the variable and value ordering
  - only sensible for variables whose values are constrained to a finite set, or the integers, etc., since we can't easily backtrack through all the infinitely many terms that might be assigned to a variable.

Constraint logic programming

- We explored at the ECLiPSe prompt or on the blackboard:
  - various small examples of CLP, e.g.,
    - X #= 2*Y, X=10.
    - X #> 2*Y, X=10.
    - member(X,[1,2,3,4,2]), X #= 2*Y.
    - X #= 2*Y, member(X,[1,2,3,4,2]).
  - uniqmember
  - simplified version of Golomb ruler (from Eclipse website)

Constraint logic programming: alldifferent

- Wrong answer:
  - alldiff([]).
  - alldiff([X|Xs]) :- member(Y,Xs), X #\= Y, alldiff(Xs).

- Right answer (although it lacks the strong propagator from ECLiPSe's standard alldifferent):
  - alldiff([]).
  - … ? (see homework)