

Logic Languages

CSE 307 – Principles of Programming Languages

Stony Brook University

<http://www.cs.stonybrook.edu/~cse307>

Languages

- Languages:
 - Imperative = Turing machines
 - Functional Programming = lambda calculus
 - Logical Programming = first-order predicate calculus
- Prolog and its variants make up the most commonly used Logical programming languages.
 - One variant is XSB → developed here at Stony Brook.
 - Prolog systems: SWI Prolog, XSB Prolog, Sicstus, Yap Prolog, Ciao Prolog, GNU Prolog, etc.
 - ISO Prolog standard.

Relations

- `parent(X, Y)`: X is a parent of Y.
 `parent(pam, bob). parent(bob, ann).`
 `parent(tom, bob). parent(bob, pat).`
 `parent(tom, liz). parent(pat, jim).`
- `male(X)`: X is a male.
 `male(tom).`
 `male(bob).`
 `male(jim).`

Relations

- `female(X)`: X is a female.

`female(pam).`

`female(pat).`

`female(ann).`

`female(liz).`

- `mother(X, Y)`: X is the mother of Y.

$\forall X, Y. \text{parent}(X; Y) \wedge \text{female}(X) \Rightarrow \text{mother}(X, Y)$

- In Prolog: `mother(X, Y) :- parent(X, Y), female(X).`

- “,” means *and* (conjunction), “:-” means *if* (implication) and “;” means *or* (disjunction).

Relations

parent(pam, bob).

parent(tom, bob).

parent(tom, liz).

parent(bob, ann).

parent(bob, pat).

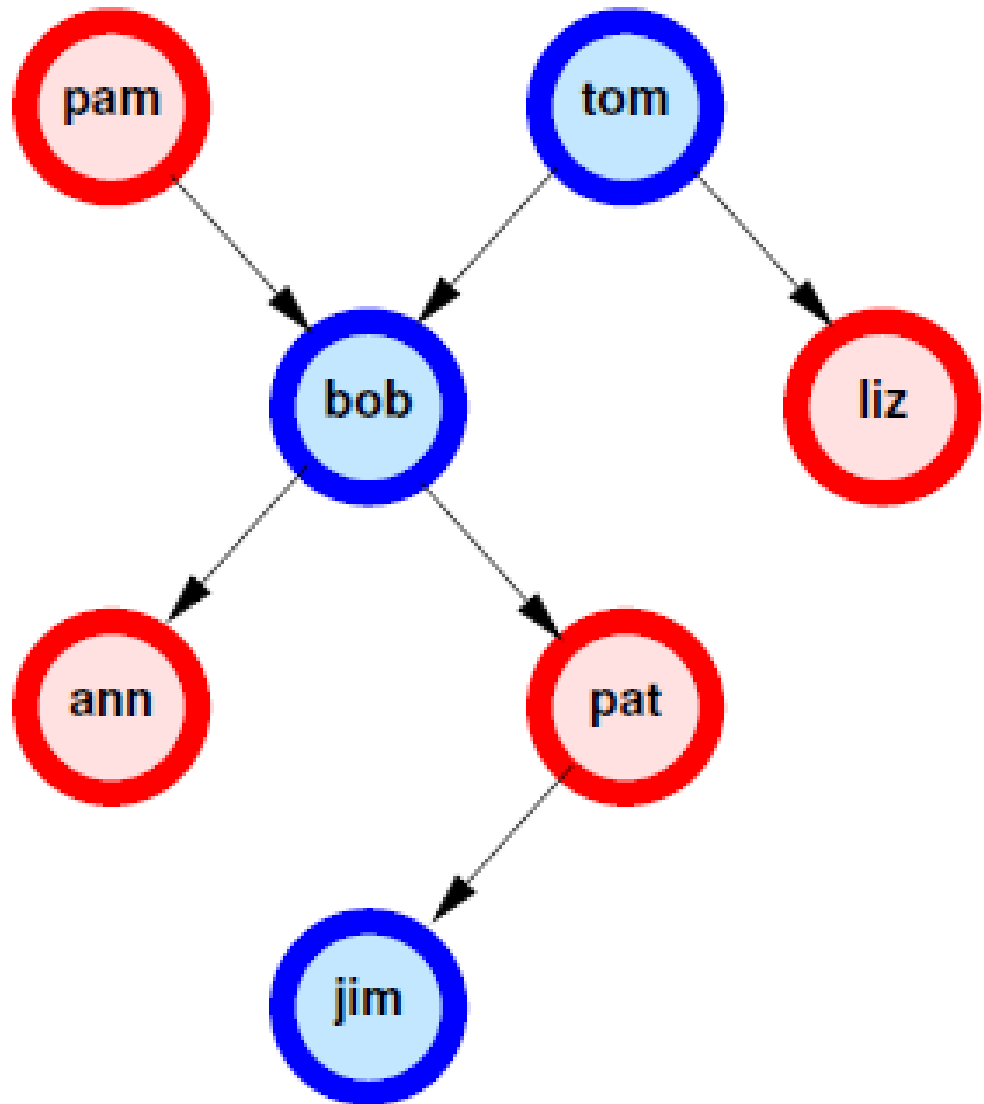
parent(pat, jim).

female(pam). male(tom).

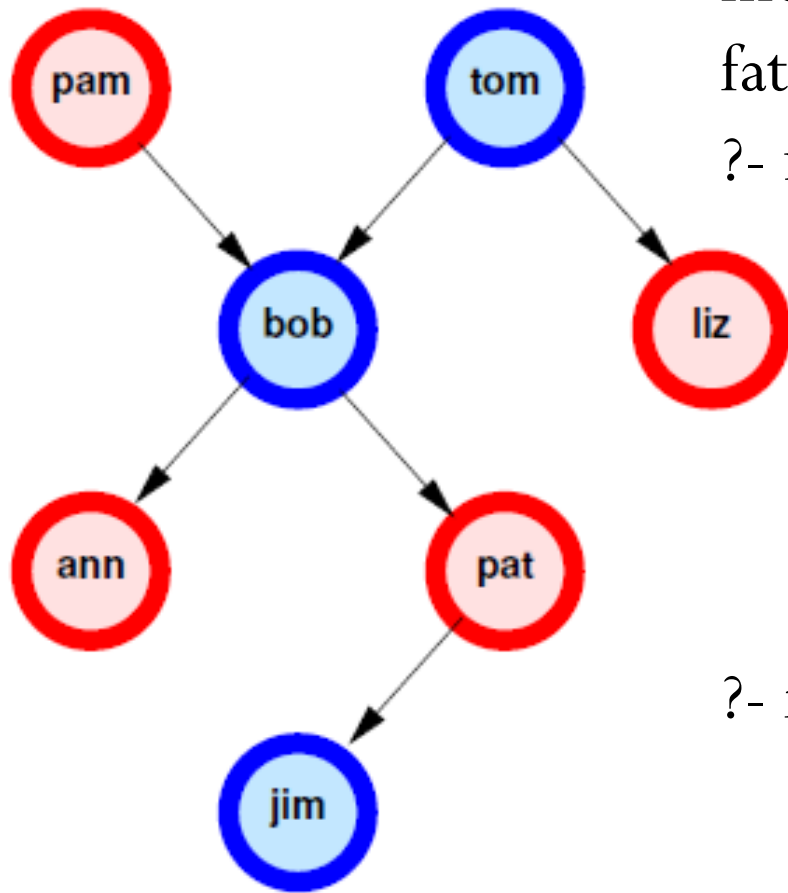
female(pat). male(bob).

female(ann). male(jim).

female(liz).



Computations in Prolog



`mother(X,Y) :- parent(X,Y), female(X).`

`father(X,Y) :- parent(X,Y), male(X).`

`?- mother(M, bob).`

|

`?- parent(M, bob), female(M).`

| `[M=pam]`

`?- female(pam).`

true

`?- father(M, bob).`

| `?- parent(M, bob), male(M)`

(i) | `?- M=pam, male(pam).`

fail

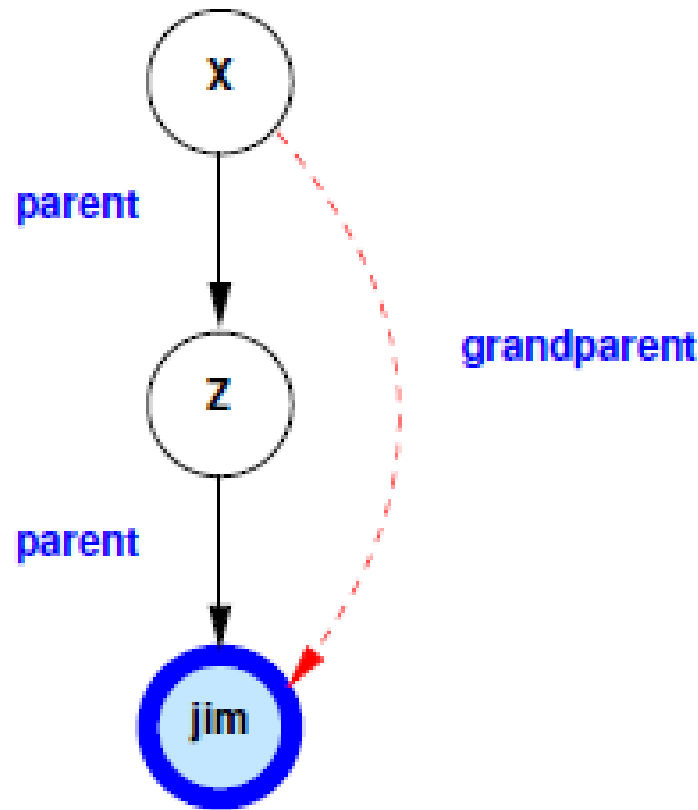
(ii) | `?- M=tom, male(tom).`

M = tom true

Relations

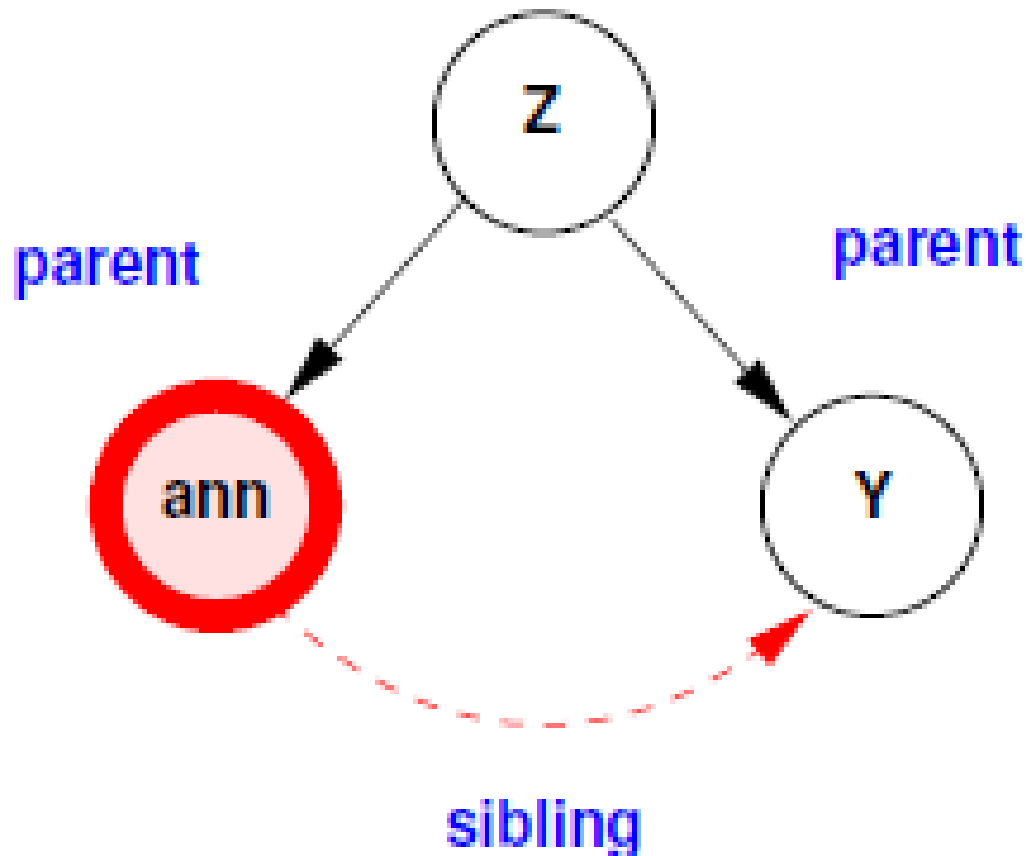
- More Relations:

$\text{grandparent}(X, Y) \text{ :- } \text{parent}(X, Z), \text{parent}(Z, Y).$



Relations

$\text{sibling}(X, Y) \text{ :- parent}(Z, X), \text{parent}(Z, Y), X \neq Y.$



Relations

- More Relations:

cousin(X,Y) :-

greatgrandparent(X,Y) :-

greatgreatgrandparent(X,Y) :-

ancestor(X,Y) :- ...

Recursion

ancestor(X,Y) :-

parent(X,Y).

ancestor(X,Y) :-

parent(X,Z),

ancestor(Z,Y).

?- ancestor(X,jim).

?- ancestor(pam,X).

?- ancestor(X,Y).

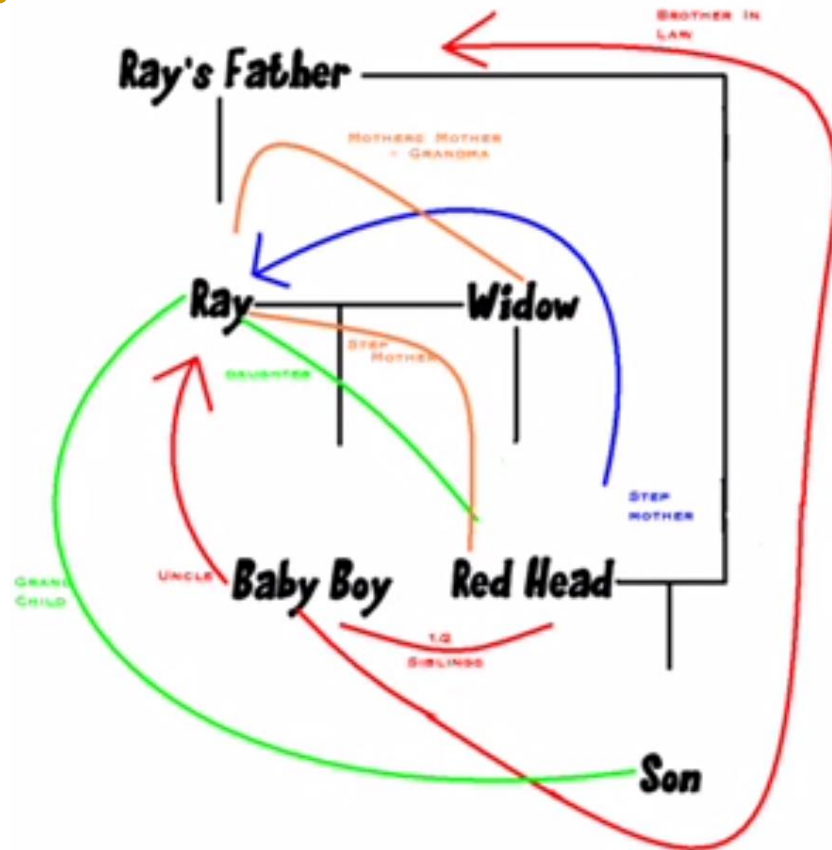
- How to implement “I'm My Own Grandpa”?

<https://www.youtube.com/watch?v=eYlJH81dSiw>

Relations

- How to implement “I’m My Own Grandpa”?

<https://www.youtube.com/watch?v=eYlJH81dSiw>



Recursion

- What about:

ancestor(X,Y) :-

 ancestor(X,Z),

 parent(Z,Y).

ancestor(X,Y) :-

 parent(X,Y).

?- ancestor(X,Y).

INFINITE LOOP

Recursion

- Transitive closure:
 - Example: a graph declared with facts (true statements)
edge(1,2).
edge(2,3).
edge(2,4).
1) if there's an edge from X to Y, we can reach Y from X.
 reachable(X,Y) :- edge(X,Y).
2) if there's an edge from X to Z, and we can reach Y from Z, we can reach Y from X.
 reachable(X,Y) :-
 edge(X,Z),
 reachable(Z,Y).

?- reachable(X,Y).

X = 1

Y = 2; ← Type a semi-colon repeatedly for

X = 2 more answers

Y = 3;

X = 2

Y = 4;

X = 1

Y = 3;

X = 1

Y = 4;

no

Prolog Execution

- Call: Call a predicate (invocation)
- Exit: Return an answer to the caller
- Fail: Return to caller with no answer
- Redo: Try next path to find an answer

Syntax of Prolog Programs

- A *program* is a sequence of clauses (Horn rules).
- Each *clause* is of the form **head :- body**.
- Head is one *term*.
- Body is a comma-separated list of terms.
- A clause with an empty body is called a *fact*.
- A clause is also sometimes called a *rule*.

Terms

- Atomic data
- Variables
- Structures

Atomic Data

- Numeric constants: Integers, floating point numbers (e.g. 1024, -42, 3.1415, 6.023e23,...)
- Atoms:
 - Strings of characters enclosed in single quotes (e.g. 'Stony Brook')
 - Identifiers: sequence of letters, digits, underscore, beginning with a letter (e.g. paul, r2d2, one_element).

Variables

- Variables are denoted by identifiers beginning with an *Uppercase letter* or *underscore* (e.g. X, Index, _param).
- Different occurrences of the same variable in a clause denote the same data.
 - Each occurrence of an anonymous variable is treated as a different data.
- Variables are implicitly declared upon first use.
- Variables are not typed.
 - All types are discovered implicitly (no declarations in LP).

Variables

- Underscore, by itself (i.e., `_`), represents an *anonymous variable*. Each occurrence of `_` corresponds to a different variable; even within a clause, `_` does not stand for one and the same object.
- *Single-variable-check*: a variable with a name beginning with a character other than `_` will be used to create relationships within a clause and must therefore be used more than once (otherwise, a warning is produced).
 - You can use variables preceded with underscore to eliminate this warning.

Variables

- Variables can be assigned only once, but that value can be further refined:

$$X=f(Y),$$

$$Y=g(Z),$$

$$Z=2.$$

- Even infinite structures: $S=f(S)$.
- We'll come to this topic later when we discuss about structures.

Logic Programming Queries

- The meaning of the statement is that *the conjunction of the terms in the body implies the head*.
 - A clause with an empty body is called a FACT: **raining(ny).**
 - A clause with both sides is a RULE: **wet(X) :- raining(X).**
X must have the same value on both sides
 - A clause with an empty head is a QUERY, or top-level GOAL:
?- wet(X).
- The Prolog interpreter has a collection of facts and rules in its DATABASE.
 - Facts are axioms - things the interpreter assumes to be true.
 - Prolog provides an automatic way to deduce true results from facts and rules.

Logic Programming Queries

- To run a Prolog program, one asks the interpreter a question
 - This is done by asking a query which the interpreter tries to prove:
 - If it can, it says yes
 - If it can't, it says no
 - If your predicate contained variables, the interpreter prints the values it had to give them to make the predicate true.

?- wet(ny).

Yes

?- wet(X).

X = ny;

X = seattle

?- reach(a, d).

Yes

?- reach(X, d).

X=a

?- reach(d, a).

No

?- reach(X, Y).

X=a, Y=d;

...

Logic Programming Rules

- Rules are theorems that allow the interpreter to infer things
- To be interesting, rules generally contain variables
employed(X) :- employs(Y,X).

can be read:

**for all X, X is employed if there
exists a Y such that Y employs X**

- The example does NOT say that X is employed
ONLY IF there is a Y that employs X

Logic Programming Rules

grandmother(A, C) :- mother(A, B), mother(B, C).

can be read:

for all A, C [A is the grandmother of C if there exists a B such that A is the mother of B and B is the mother of C].

- We want another rule that says

grandmother(A, C) :- mother(A, B), father(B, C).

The XSB Prolog System

- <http://xsb.sourceforge.net>
 - Developed at Stony Brook by David Warren and many contributors.
- Overview of Installation:
 - Unzip/untar; this will create a subdirectory XSB
 - Windows: you are done
 - Linux:
 - `cd XSB/build`
 - `./configure`
 - `./makexsb`
 - That's it!
 - Cygwin under Windows: same as in Linux

Use of XSB

- Put your ruleset *and* data in a file with extension .P (or .pl)
 $p(X) :- q(X, _).$
 $q(1, a).$
 $q(2, a).$
 $q(b, c).$
 $?- p(X).$
- Don't forget: all rules and facts end with a period (.)
- Comments: `/*...*/` or `%....` (% acts like `//` in Java/C++)

- Type

.../XSB/bin/xsb (Linux/Cygwin)

...\XSB\config\x86-pc-windows\bin\xsb (Windows)

where ... is the path to the directory where you downloaded XSB

- You will see a prompt

| ? -

and are now ready to type queries

Use of XSB

- Loading your program, myprog.P

| ?- [myprog].

XSB will compile myprog.P (if necessary) and load it. Now you can type further queries, e.g.

| ?- p(X).

| ?- p(1).

Etc.

- Some Useful Built-ins:

- write(X) – write whatever X is bound to
- writeln(X) – write then put newline
- nl – output newline
- Equality: =
- Inequality: \=

<http://xsb.sourceforge.net/manual1/index.html> (Volume 1)

<http://xsb.sourceforge.net/manual2/index.html> (Volume 2)

(c) Paul Fodor (CS Stony Brook) and Elsevier

Use of XSB

- Some Useful Tricks:

- XSB returns only the first answer to the query. To get the next, type `; <Return>`. For instance:

```
| ?- q(X).
```

```
X = 2;
```

```
X = 4
```

```
yes
```

```
| ?-
```

- Usually, typing the `;`'s is tedious. To do this programmatically, use this idiom:

```
| ?- (q(_X), write('X='), writeln(_X), fail ; true).
```

`_X` here tells XSB to not print its own answers, since we are printing them by ourselves. (XSB won't print answers for variables that are prefixed with a `_`.)

Meaning of Logic Programs

- **Declarative Meaning:** What are the logical consequences of a program?
- **Procedural Meaning:** For what values of the variables in the query can I *prove* the query?
- The user gives the system a goal:
 - The system attempts to find axioms + inference steps to prove goal.
 - If goal contains variables, then also gives the values for those variables.

Declarative Meaning

```
brown(bear) .           big(bear) .  
gray(elephant) .      big(elephant) .  
black(cat) .          small(cat) .  
dark(Z) :- black(Z) .  
dark(Z) :- brown(Z) .  
dangerous(X) :- dark(X) , big(X) .
```

- *Logical consequence of a program* L is the smallest set such that
 - All facts of the program are in L,
 - If $H :- B_1, B_2, \dots, B_n$ is an instance of a clause in the program such that B_1, B_2, \dots, B_n are all in L, then H is also in L.
- For the above program we get `dark(cat)` and `dark(bear)` and consequently `dangerous(bear)`.

Procedural Meaning of Prolog

- The Prolog interpreter works by what is called BACKWARD CHAINING (top-down, goal directed)
 - It begins with the thing it is trying to prove and works backwards looking for things that would imply it, until it gets to facts.
- It is also possible in theory to work forward from the facts trying to see if any of the things you can prove from them are what you were looking for (bottom-up resolution) - that can be very time-consuming
 - Example: Answer set programming, DLV, Potassco (the Potsdam Answer Set Solving Collection), OntoBroker
 - Fancier logic languages use both kinds of chaining, with special smarts or hints from the user to bound the searches

Procedural Meaning of Prolog

- The interpreter starts at the beginning of your database (this ordering is part of Prolog, NOT of logic programming in general) and looks for something with which to unify the current goal
 - If it finds a fact, great; it succeeds,
 - If it finds a rule, it attempts to satisfy the terms in the body of the rule depth first.
- This process is motivated by the RESOLUTION PRINCIPLE, due to Robinson:
 - It says that if $C1$ and $C2$ are Horn clauses, where $C2$ represents a true statement and the head of $C2$ unifies with one of the terms in the body of $C1$, then we can replace the term in $C1$ with the body of $C2$ to obtain another statement that is true if and only if $C1$ is true

Procedural Meaning of Prolog

- When it attempts resolution, the Prolog interpreter pushes the current goal onto a stack, makes the first term in the body the current goal, and goes back to the beginning of the database and starts looking again.
- If it gets through the first goal of a body successfully, the interpreter continues with the next one.
- If it gets all the way through the body, the goal is satisfied and it backs up a level and proceeds.

Procedural Meaning of Prolog

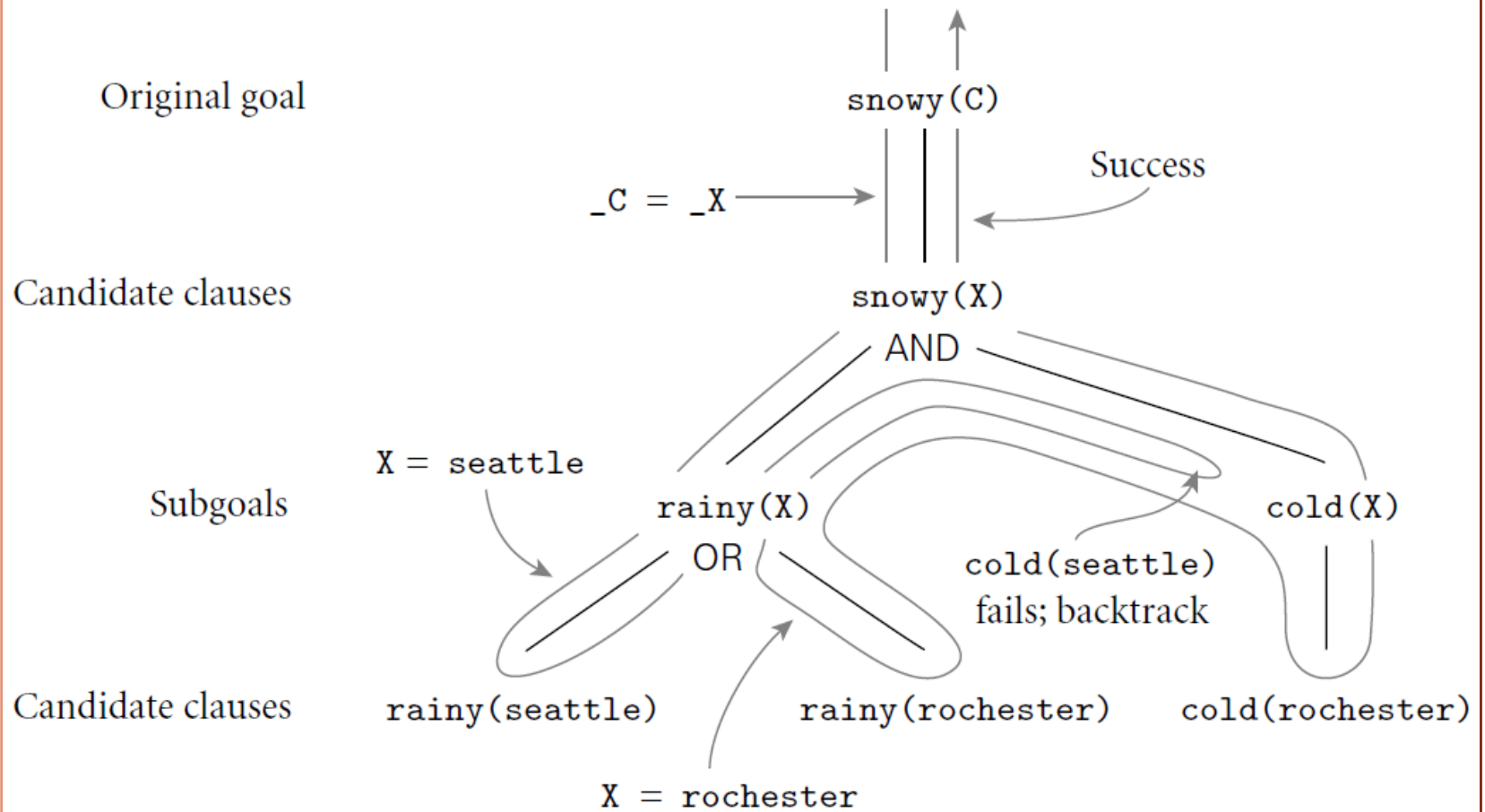
- If it fails to satisfy the terms in the body of a rule, the interpreter undoes the unification of the left hand side (this includes uninstantiating any variables that were given values as a result of the unification) and keeps looking through the database for something else with which to unify (This process is called BACKTRACKING).
- If the interpreter gets to the end of database without succeeding, it backs out a level (that's how it might fail to satisfy something in a body) and continues from there.

Procedural Meaning of Prolog

- We can visualize backtracking search as a tree in which the top-level goal is the root and the leaves are facts
 - The children of the root are all the rules and facts with which the goal can unify
 - The interpreter does an OR across them: one of them must succeed in order for goal to succeed
 - The children of a node in the second level of the tree are the terms in the body of the rule
 - The interpreter does an AND across these: all of them must succeed in order for parent to succeed
 - The overall search tree then consists of alternating AND and OR levels

Procedural Meaning of Prolog

```
rainy(seattle).  
rainy(rochester).  
cold(rochester).  
snowy(X) :- rainy(X), cold(X).
```



Procedural Meaning of Prolog

```
brown(bear) .           big(bear) .
gray(elephant) .       big(elephant) .
black(cat) .           small(cat) .
dark(Z) :- black(Z) .
dark(Z) :- brown(Z) .
dangerous(X) :- dark(X) , big(X) .
```

- A *query* is, in general, a conjunction of goals: G_1, G_2, \dots, G_n
- To *prove* G_1, G_2, \dots, G_n :
 - Find a clause $H :- B_1, B_2, \dots, B_k$ such that G_1 and H match.
 - Under that substitution for variables, prove $B_1, B_2, \dots, B_k, G_2, \dots, G_n$

If nothing is left to prove then the proof succeeds!

If there are no more clauses to match, the proof fails!

Procedural Meaning of Prolog

```
brown(bear) .           big(bear) .
gray(elephant) .       big(elephant) .
black(cat) .           small(cat) .
dark(Z) :- black(Z) .
dark(Z) :- brown(Z) .
dangerous(X) :- dark(X) , big(X) .
```

- To prove: ?- **dangerous(Q)** .

1. Select **dangerous(X) :- dark(X), big(X)** and prove **dark(Q), big(Q)**.
2. To prove **dark(Q)** select the first clause of dark, i.e. **dark(Z) :- black(Z)**, and prove **black(Q), big(Q)**.
3. Now select the fact **black(cat)** and prove **big(cat)**. **This proof fails!**
4. Go back to step 2, and select the second clause of dark, i.e. **dark(Z) :- brown(Z)**, and prove **brown(Q), big(Q)**.

Procedural Meaning of Prolog

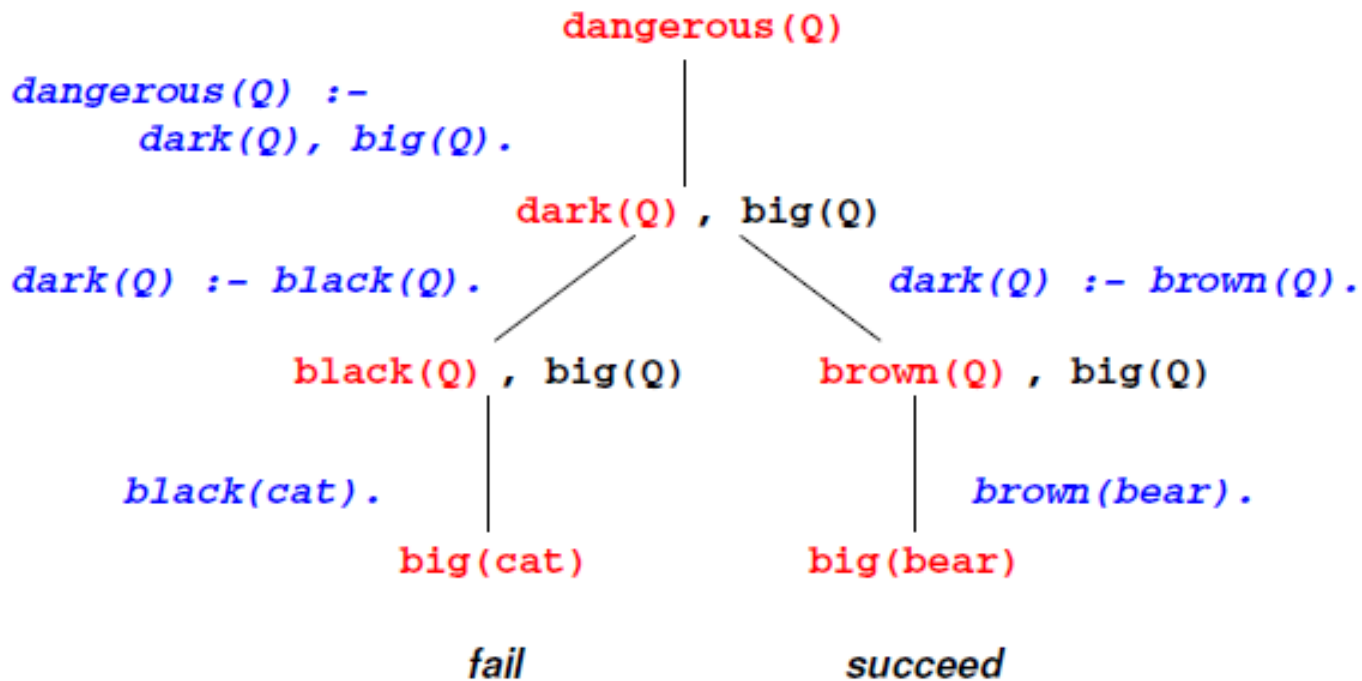
```
brown(bear) .           big(bear) .  
gray(elephant) .       big(elephant) .  
black(cat) .           small(cat) .  
dark(Z) :- black(Z) .  
dark(Z) :- brown(Z) .  
dangerous(X) :- dark(X) , big(X) .
```

- To prove: **?- dangerous(Q) .**
 5. Now select **brown(bear)** and prove **big(bear)**.
 6. Select the fact **big(bear)**.

There is nothing left to prove, so the proof succeeds

Procedural Meaning of Prolog

```
brown(bear) .           big(bear) .  
gray(elephant) .       big(elephant) .  
black(cat) .           small(cat) .  
dark(Z) :- black(Z) .  
dark(Z) :- brown(Z) .  
dangerous(X) :- dark(X) , big(X) .
```

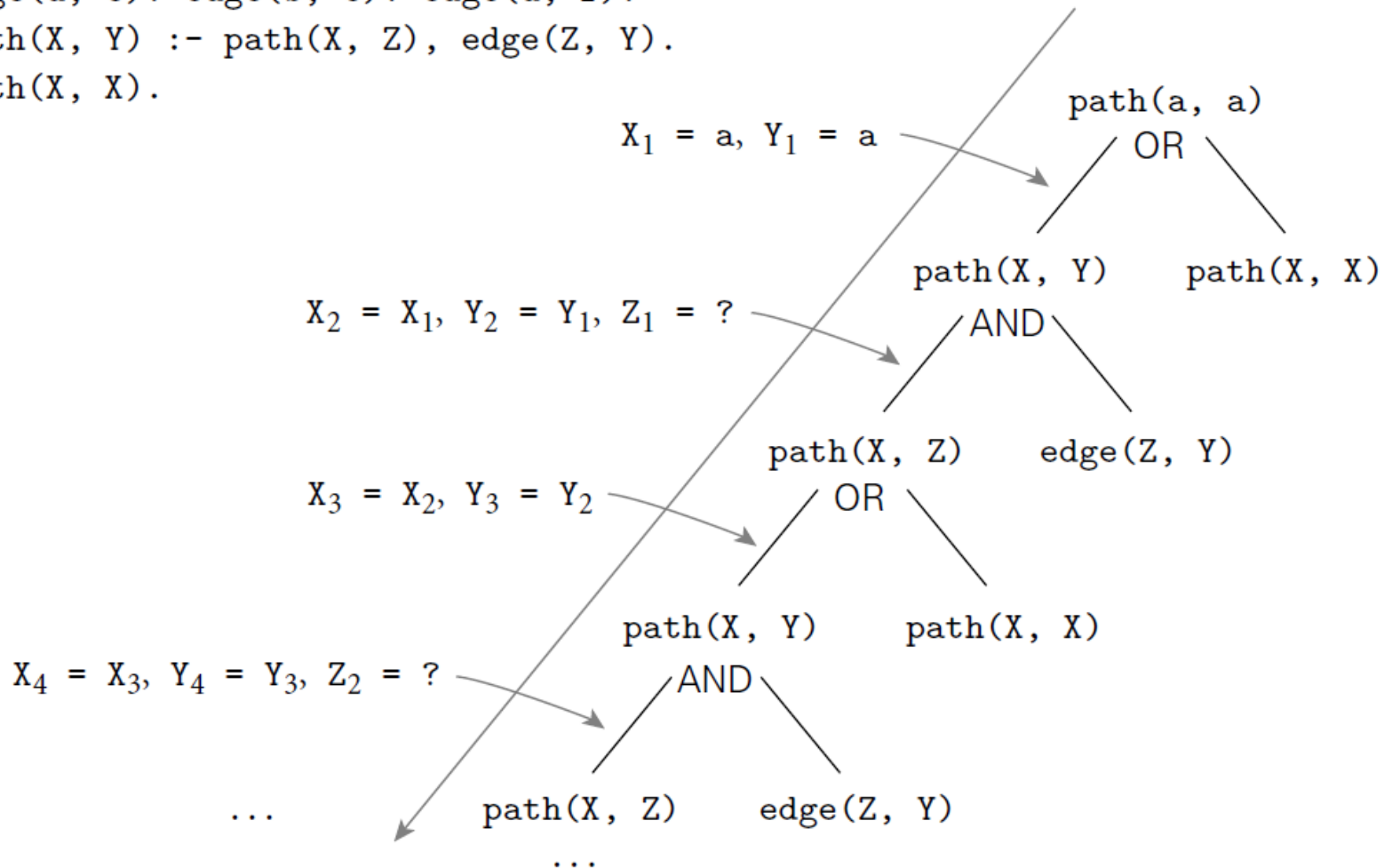


Prolog

- PROLOG IS NOT PURELY DECLARATIVE:
 - The ordering of the database and the left-to-right pursuit of sub-goals gives a deterministic imperative semantics to searching and backtracking,
 - Changing the order of statements in the database can give you different results:
 - It can lead to infinite loops,
 - It can certainly result in inefficiency.

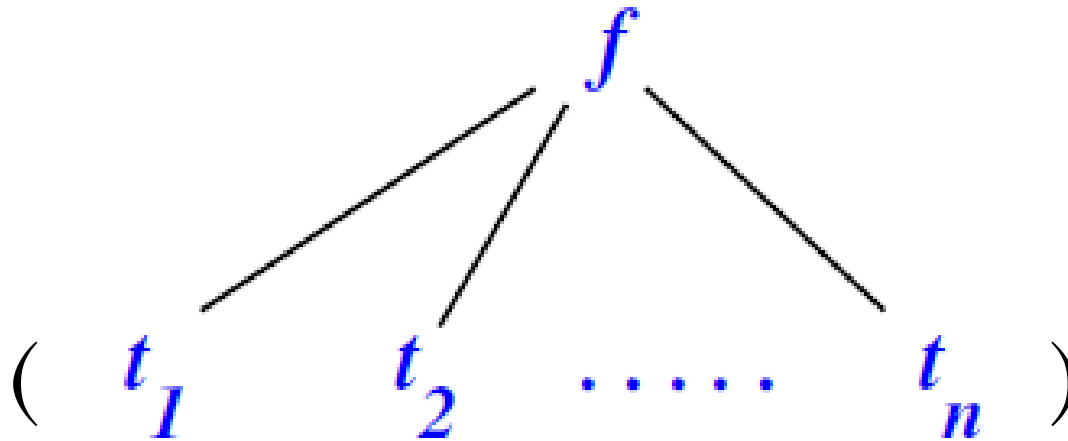
Infinite regression in Prolog

```
edge(a, b). edge(b, c). edge(c, d).  
edge(d, e). edge(b, e). edge(d, f).  
path(X, Y) :- path(X, Z), edge(Z, Y).  
path(X, X).
```



Structures

- If f is an identifier and t_1, t_2, \dots, t_n are terms, then $f(t_1, t_2, \dots, t_n)$ is a term.

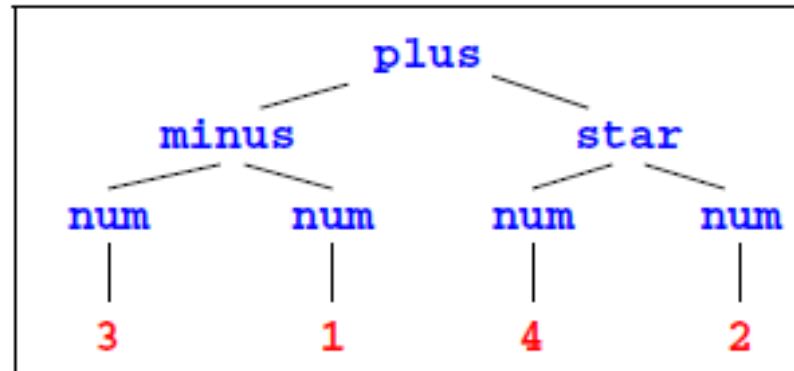


- In the above, f is called a *functor* and t_i is an *argument*.
- Structures are used to group related data items together (in some ways similar to struct in C and objects in Java).
- Structures are used to construct trees (and, as a special case, lists).

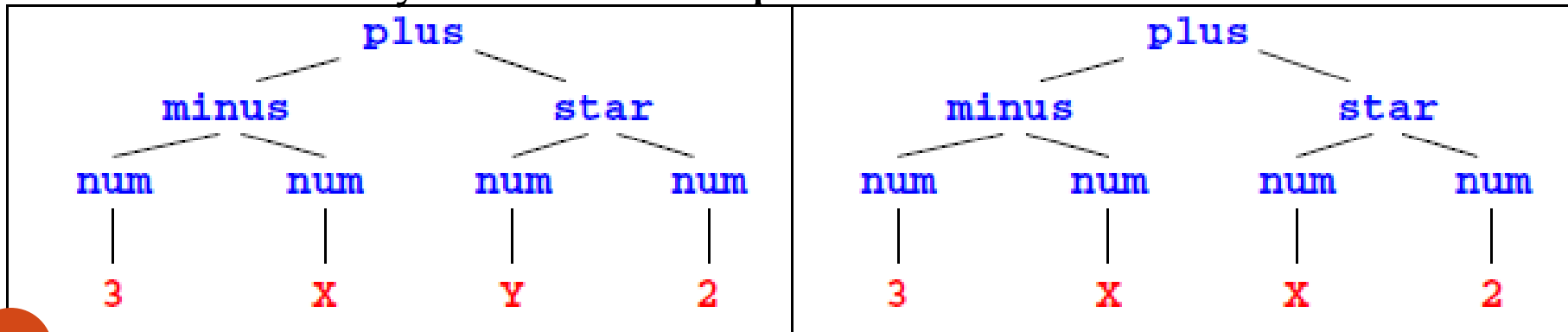
Trees

- Example: expression trees:

`plus (minus (num (3) , num (1)) , star (num (4) , num (2)))`

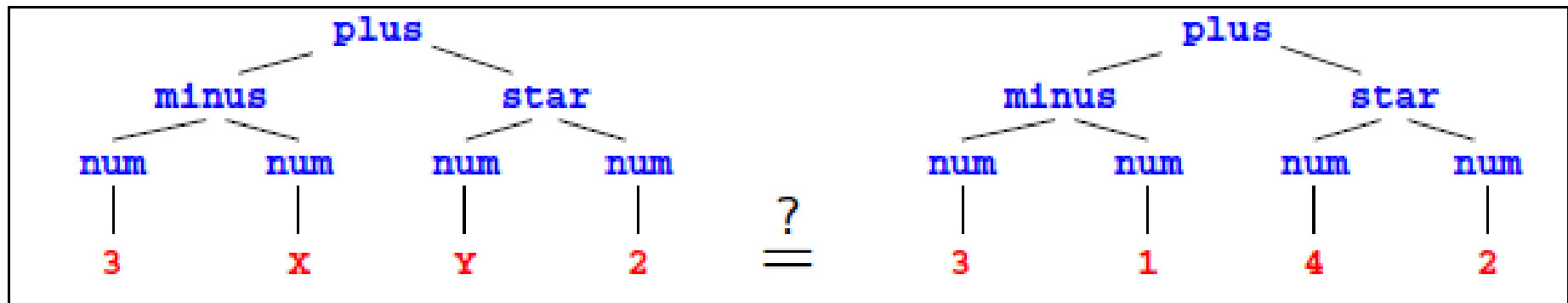


- Data structures may have variables. And the same variable may occur multiple times in a data structure.



Matching

- (We'll later introduce *unification*, a related operation that has logical semantics).
- $t_1 = t_2$: find substitutions for variables in t_1 and t_2 that make the two terms identical.

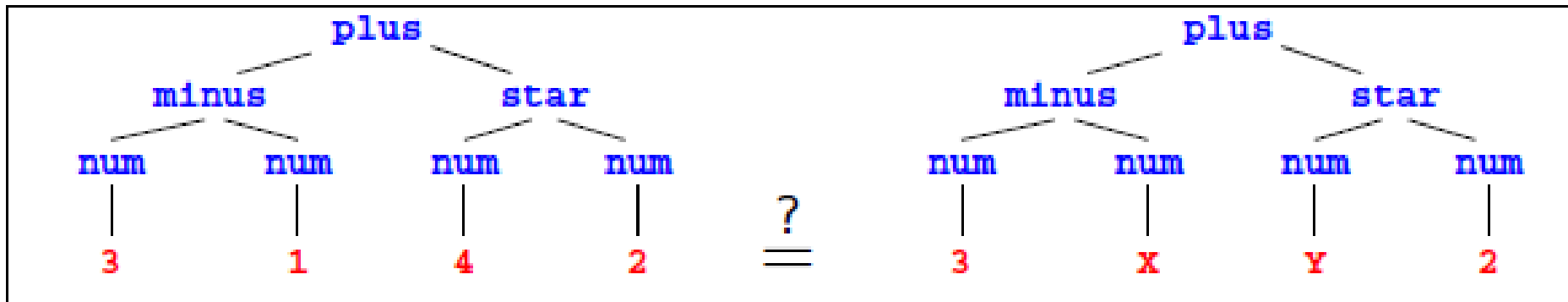


Yes, with $X = 1$, $Y = 4$.

Matching

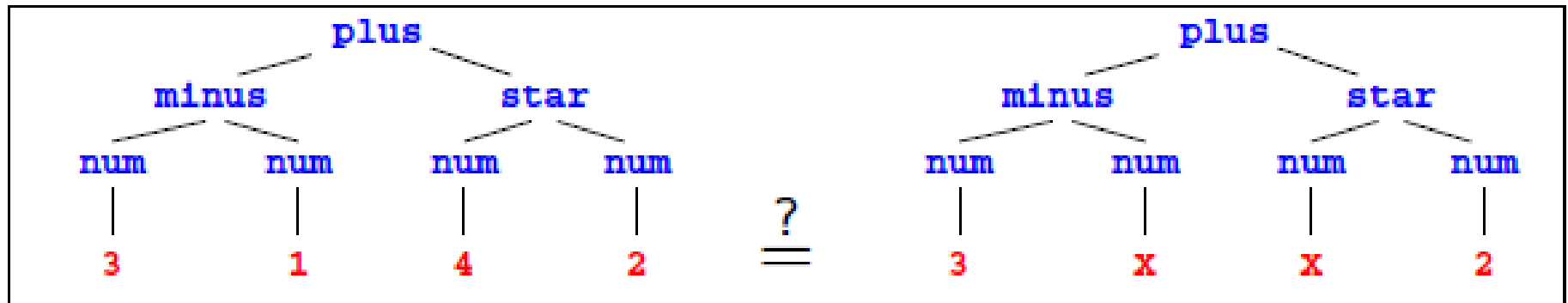
- Matching: given two terms, we can ask if they "match" each other. Rules:
 - A constant matches with itself: 42 unifies with 42.
 - A variable matches with anything:
 - if it matches with something other than a variable, then it instantiates,
 - if it matches with a variable, then the two variables become associated.
 - $A=35, A = B \rightarrow B$ becomes 35.
 - $A = B, A=35 \rightarrow B$ becomes 35.
 - Two structures match if they:
 - Have the same functor,
 - Have the same arity,
 - Match recursively.
 - $\text{foo}(\text{g}(42), 37)$ matches with $\text{foo}(A, 37)$, $\text{foo}(\text{g}(A), B)$, etc.

Matching



Yes, with $X = 1$, $Y = 4$.

Matching



No! X cannot be 1 and 4 at the same time.

Matching

- Which of these match?
 - A
 - 100
 - func(B)
 - func(100)
 - func(C, D)
 - func(+ (99, 1))

A matches with 100, func(B), func(100), func(C,D), func(+ (99, 1)).

100 matches with A.

func(B) matches with A, func(100), func(+ (99, 1))

func(C, D) matches with A.

func(+ (99, 1)) matches with A, func(B).

Accessing arguments of a structure

- Matching is the predominant means for accessing a structures arguments.
- Let `date('Sep', 1, 2015)` be a structure used to represent dates, with the month, day and year as the three arguments (**in that order!**).

Then `date(M, D, Y) = date('Sep', 1, 2015)` makes

`M = 'Sep', D = 1, Y = 2015.`

- If we want to get only the day, we can write `date(_, D, _) = date('Sep', 1, 2015).`

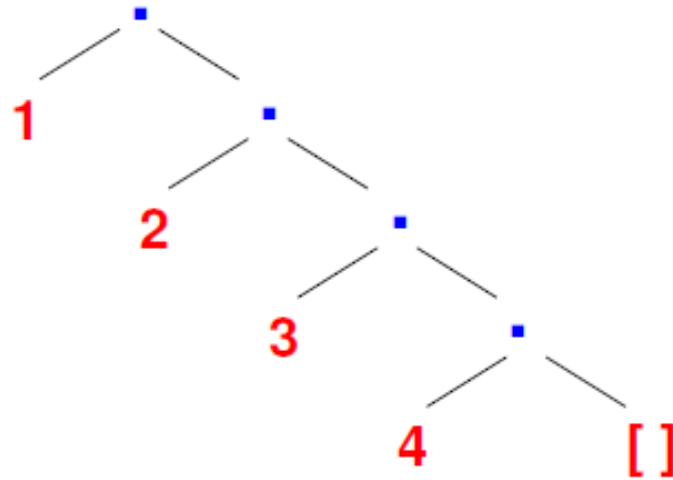
Then we only get: `D = 1.`

Lists

- Prolog uses a special syntax to represent and manipulate lists (syntactic sugar = internally, it uses structures):
 - $[1,2,3,4]$: represents a list with 1, 2, 3 and 4, respectively.
 - This can also be written as $[1 \mid [2,3,4]]$: a list with 1 as the *head* (first element) and $[2,3,4]$ as its *tail* (the list of remaining elements).
 - If $X = 1$ and $Y = [2,3,4]$ then $[X \mid Y]$ is same as $[1,2,3,4]$.
 - The empty list is represented by $[]$ or *nil*.
 - The symbol " \mid " (*pipe*) and is used to separate the beginning elements of a list from its tail.
 - For example: $[1,2,3,4] = [1 \mid [2,3,4]] = [1 \mid [2 \mid [3,4]]] = [1,2 \mid [3,4]]$

Lists

- Lists are special cases of trees (i.e., (syntactic sugar = internally, it uses structures).
- For instance, the list $[1,2,3,4]$ is represented by the following structure:



- where the function symbol $./2$ is the list constructor.
 $[1,2,3,4]$ is same as $.(1, .(2, .(3, .(4, []))))$

Lists

- *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes: “abc”, “to be, or not to be”.

Programming with Lists

- First example: `member/2`, to find if a given element occurs in a list:

- The program:

```
member (X, [X|_]) .
```

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

- Example queries:

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

```
?- member (X, [1,i,s,t]) .
```

```
?- member (f(X), [f(1),g(2),f(3),h(4)]) .
```

Programming with Lists

- append/3: concatenate two lists to form the third list:
- The program:

- Empty list append A is A.

append([], L, L) .

- Otherwise, break the first list up into a head X, tail L: if L append M is N, then X|N append M is X|N:

**append([X|L], M, [X|N]) :-
append(L, M, N) .**

- Example queries:

?- append([1,2], [3,4], X) .

?- append(X, Y, [1,2,3,4]) .

?- append(X, [3,4], [1,2,3,4]) .

Programming with Lists

- Is the predicate a function?
 - No. We are not applying arguments to get a result. Instead, we are proving that a theorem holds. Therefore, we can leave other variables unbound.

?- append(L, [2, 3], [1, 2, 3]). L = [1]

?- append([1], L, [1, 2, 3]). L = [2, 3]

?- append(L1, L2, [1, 2, 3]).

L1 = [] L2 = [1, 2, 3];

L1 = [1] L2 = [2, 3];

L1 = [1, 2] L2 = [3] ;

L1 = [1, 2, 3] L2 = [];

no

Append example

```
append([], L, L) .
```

```
append([X|L], M, [X|N]) :- append(L, M, N) .
```

```
append([1,2], [3,4], X) ?
```

Append example

`append([], L, L) .`

`append([X|L], M, [X|N]) :- append(L, M, N) .`

<code>append([1, 2], [3, 4], A) ?</code>	<code>X=1, L=[2], M=[3, 4], A=[X N]</code>
--	--

Append example

`append([], L, L) .`

`append([X|L], M, [X|N]) :- append(L, M, N) .`

`append([2], [3, 4], N) ?`

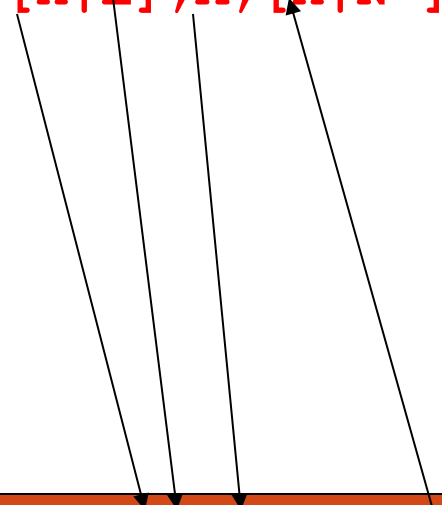
`append([1, 2], [3, 4], A) ?`

`X=1, L=[2], M=[3, 4], A=[X|N]`

Append example

`append([], L, L) .`

`append([X|L], M, [X|N']) :- append(L, M, N') .`



<code>append([2], [3, 4], N) ?</code>	<code>X=2, L=[], M=[3, 4], N=[2 N']</code>
<code>append([1, 2], [3, 4], A) ?</code>	<code>X=1, L=[2], M=[3, 4], A=[1 N]</code>

Append example

append([] , L , L) .

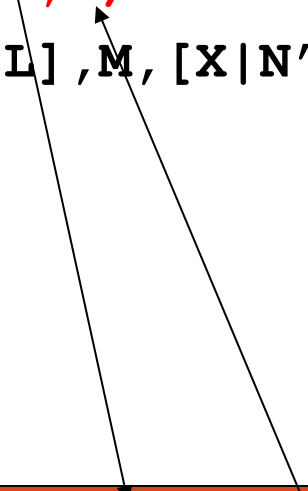
append([X|L] , M , [X|N']) :- **append**(L , M , N') .

append ([] , [3,4] , N') ?	
append ([2] , [3,4] , N) ?	X=2 , L= [] , M= [3,4] , N= [2 N']
append ([1,2] , [3,4] , A) ?	X=1 , L= [2] , M= [3,4] , A= [1 N]

Append example

append([], L, L) .

append([X|L], M, [X|N']) :- append(L, M, N') .



append([], [3,4], N') ?	L = [3,4], N' = L
append([2], [3,4], N) ?	X=2, L=[], M=[3,4], N=[2 N']
append([1,2], [3,4], A) ?	X=1, L=[2], M=[3,4], A=[1 N]

Append example

`append([], L, L) .`

`append([X|L], M, [X|N']) :- append(L, M, N') .`

`A = [1|N]`

`N = [2|N']`

`N' = L`

`L = [3,4]`

Answer: `A = [1,2,3,4]`

`append([], [3,4], N') ?`

`L = [3,4], N' = L`

`append([2], [3,4], N) ?`

`X=2, L=[], M=[3,4], N=[2|N']`

`append([1,2], [3,4], A) ?`

`X=1, L=[2], M=[3,4], A=[1|N]`

Programming with Lists

- `len/2` finds the length of a list (first argument).:

- The program:

```
len([], 0) .
```

```
len([_|Xs], N+1) :- len(Xs, N) .
```

- Example queries:

```
?- len([], X) .
```

```
?- len([l,i,s,t], 4) .
```

```
?- len([l,i,s,t], X) .
```

Arithmetic

?- $1 + 2 = 3$.

no

- In Predicate logic, the basis for Prolog, the only symbols that have a meaning are the predicates themselves.
- In particular, function symbols are uninterpreted: have no special meaning and can only be used to construct data structures.

Arithmetic

- Meaning for arithmetic expressions is given by the built-in predicate "is":

?- X is 1 + 2.

succeeds, binding X to 3.

?- 3 is 1 + 2.

succeeds.

- General form: R is E where E is an expression to be evaluated and R is matched with the expression's value.
- Y is X + 1, where X is a free variable, will give an error because X does not (yet) have a value, so, X + 1 cannot be evaluated.

The list length example revisited

- `length/2` finds the length of a list (first argument).:
- The program:

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

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

- Example queries:

```
?- length([], X) .
```

```
?- length([l,i,s,t], 4) .
```

```
?- length([l,i,s,t], X) .
```

```
?- length(List, 4) .
```

Conditional Evaluation

- Consider the computation of $n!$ (i.e. the factorial of n)

factorial(N, F) :- ...

- N is the input parameter; and F is the output parameter!
- The body of the rule specifies how the output is related to the input.
- For factorial, there are two cases: $N \leq 0$ and $N > 0$.
 - if $N \leq 0$, then $F = 1$
 - if $N > 0$, then $F = N * \text{factorial}(N - 1)$

factorial(N, F) :-

(N > 0

-> N1 is N-1, factorial(N1, F1), F is N*F1

; F = 1

).

Conditional Evaluation

- Conditional operator: the if-then-else construct in Prolog:
 - *if A then B else C* is written as $(A \text{ -> } B ; C)$
 - To Prolog this means: try A. If you can prove it, go on to prove B and ignore C. If A fails, however, go on to prove C ignoring B.

```
max (X, Y, Z) :-  
    ( X =< Y  
    -> Z = Y  
    ; Z = X  
    ) .
```

Imperative features

- Other imperative features: we can think of prolog rules as imperative programs w/ backtracking.

program :-

member(X, [1, 2, 3, 4]),

write(X),

nl,

fail.

program.

- Fail: always fails, causes backtracking.
- ! is the cut operator: prevents other rules from matching (see Cut lecture note).

Therefore, Prolog Syntax:

- Assignments with arithmetic expressions is done using the keyword "is".
- If-then-else is written as
$$(\text{ cond } \rightarrow \text{ then-part } ; \text{ else-part })$$
- If more than one action needs to be performed in a rule, they are written one after another, separated by a comma.
- Arithmetic expressions are not directly used as arguments when calling a predicate; they are first evaluated, and then passed to the called predicate.

Arithmetic Operators

- Integer/Floating Point operators: $+$, $-$, $*$, $/$
 - Automatic detection of Integer/Floating Point
- Integer operators: mod , $//$ (div)
- Int \leftrightarrow Float operators: floor, ceiling
- Comparison operators: $<$, $>$, $=<$, $>=$,
 $\text{Expr1} ::= \text{Expr2}$ (succeeds if expression
 Expr1 evaluates to a number equal to Expr2),
 $\text{Expr1} = \backslash = \text{Expr2}$ (succeeds if expression
 Expr1 evaluates to a number non-equal to Expr2)

Programming with Lists

- Define `delete/3`, to remove a given element from a list (called `select/3` in XSB's basics library):
 - E.g. `delete([1,2,3], 2, X)` should succeed with `X = [1,3]`.
- *When X is selected from $[X \mid Ys]$, Ys results.*
- *When X is selected from the tail of $[X \mid Ys]$, $[X \mid Zs]$ results, where Zs is the result of taking X out of Ys .*

Programming with Lists

- E.g. `delete([1,2,3], 2, X)` should succeed with `X = [1,3]`.

- The program:

```
delete([X|Ys], X, Ys).
```

```
delete([Y|Ys], X, [Y|Zs]) :-  
    delete(Ys, X, Zs).
```

- Example queries:

```
?- delete([l,i,s,t], s, X).
```

```
?- delete([l,i,s,t], X, Y).
```

```
?- delete(X, s, [l,i,t]).
```

```
?- delete(X, Y, [l,i,s,t]).
```

Permutations

- Define `permute/2`, to find a permutation of a given list.
 - E.g. `permute([1,2,3], X)` should return `X=[1,2,3]` and upon backtracking, `X=[1,3,2]`, `X=[2,1,3]`, `X=[2,3,1]`, `X=[3,1,2]`, and `X=[3,2,1]`.
 - Hint: What is the relationship between the permutations of `[1,2,3]` and the permutations of `[2,3]`?

<code>permute([2,3], Y)</code>	<code>permute([1,2,3], Y)</code>
<code>[2,3]</code>	<code>[1,2,3]</code>
	<code>[2,1,3]</code>
	<code>[2,3,1]</code>
<code>[3,2]</code>	<code>[1,3,2]</code>
	<code>[3,1,2]</code>
	<code>[3,2,1]</code>

Programming with Lists

- The program:

```
permute([], []).
```

```
permute([X|Xs], Ys) :-
```

```
    permute(Xs, Zs),
```

```
    delete(Ys, X, Zs).
```

- Example query:

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

The Issue of Efficiency

- Define a predicate, `rev/2` that finds the reverse of a given list.
 - E.g. `rev([1,2,3], X)` should succeed with $X = [3,2,1]$.
 - Hint: what is the relationship between the reverse of `[1,2,3]` and the reverse of `[2,3]`?

`rev([], []).`

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

- How long does it take to evaluate `rev([1, 2, ..., n], X)`?
 - $T(n) = T(n - 1) + \text{time to add 1 element to the end of an } n - 1 \text{ element list}$
 $= T(n - 1) + n - 1 = T(n - 2) + n - 2 + n - 1 = \dots$
 - $\rightarrow T(n) = O(n^2)$

Making rev/2 faster

- Keep an accumulator: a stack all elements seen so far.
 - i.e. a list, with elements seen so far in reverse order.

- The program:

```
rev(L1, L2) :- rev(L1, [], L2).
```

```
rev([X|Xs], AccBefore, AccAfter) :-  
    rev(Xs, [X|AccBefore], AccAfter).
```

```
rev([], Acc, Acc). % Base case
```

- Example query:

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

```
which calls rev([2,3], [1], X)
```

```
which calls rev([3], [2,1], X)
```

```
which calls rev([], [3,2,1], X)
```

Tree Traversal

- Assume you have a binary tree, represented by
 - node/ 3 facts: for internal nodes: `node(a,b,c)` means that a has b and c as children.
 - leaf/ 1 facts: for leaves: `leaf(a)` means that a is a leaf.
 - Example:
`node(5, 3, 6). node(3, 1, 4). leaf(1). leaf(4). leaf(6).`
- Write a predicate `preorder/ 2` that traverses the tree (starting from a given node) and returns the list of nodes in pre-order

Tree Traversal

```
preorder(Root, [Root]) :- leaf(Root).  
preorder(Root, [Root|L]) :-  
    node(Root, Child1, Child2),  
    preorder(Child1, L1),  
    preorder(Child2, L2),  
    append(L1, L2, L).
```

- The program takes $O(n^2)$ time to traverse a tree with n nodes.

Difference Lists

- The lists in Prolog are singly-linked; hence we can access the first element in constant time, but need to scan the entire list to get the last element.
- However, unlike functional languages like Lisp or SML, we can use variables in data structures:
 - We can exploit this to make lists “open tailed”

Difference Lists

- When $X = [1, 2, 3 \mid Y]$, X is a list with 1, 2, 3 as its first three elements, followed by Y .
 - Now if $Y = [4 \mid Z]$ then $X = [1, 2, 3, 4 \mid Z]$.
 - We can think of Z as “pointing to” the end of X .
 - **We can now add an element to the end of X in constant time!!**
 - (e.g. $Z = [5 \mid W]$)
- Open-tailed lists are also called *difference lists* in Prolog.

Tree Traversal, Revisited

```
preorder1(Node, List, Tail) :-  
    node(Node, Child1, Child2),  
    List = [Node|List1],  
    preorder1(Child1, List1, Tail1),  
    preorder1(Child2, Tail1, Tail).  
preorder1(Node, [Node|Tail], Tail) :-  
    leaf(Node).  
preorder(Node, List) :-  
    preorder1(Node, List, []).
```

- The program takes $O(n)$ time to traverse a tree with n nodes.

Difference Lists: Conventions

(Chap. 8.5.3 of Bratko)

- An difference list is represented by two variables: one referring to the entire list, and another to its (uninstantiated) tail.
 - e.g. $X = [1, 2, 3 \mid Z]$.
- Most Prolog programmers use the notation List - Tail to denote a list List with tail Tail.
- Note that “-” is used as a data structure symbol (not used here for arithmetic).

Difference Lists: Conventions

- The preorder traversal program may be written as:

```
preorder1 (Node, [Node|L]-T) :-  
    node (Node, Child1, Child2),  
    preorder1 (Child1, L-T1),  
    preorder1 (Child2, T1-T).  
preorder1 (Node, [Node|T]-T).
```

Graphs in Prolog

- There are several ways to represent graphs in Prolog:
 - represent each edge separately as one clause (fact):
edge(a,b).
edge(b,c). ...
 - isolated nodes cannot be represented, unless we have also node/1 facts
 - the whole graph as one data object: as a pair of two sets (nodes and edges): `graph([a,b,c,d,f,g],[e(a,b), e(b,c),e(b,f)])`
 - list of arcs: [a-b, b-c, b-f]
 - adjacency-list: [n(a,[b]), n(b,[c,f]), n(d,[])]

Graphs in Prolog

- Path from one node to another one:
 - a predicate `path(G,A,B,P)` to find an acyclic path `P` from node `A` to node `B` in the graph `G`.
 - The predicate should return all paths via backtracking.
 - We will solve it using the graph as a data object, like in `graph([a,b,c,d,f,g],[e(a,b), e(b,c),e(b,f)])`

Graphs in Prolog

- Path from one node to another one:

```
path(G,A,B,P) :- path1(G,A,[B],P).
```

```
path1( _,A,[A | P1],[A | P1]).
```

```
path1(G,A,[Y | P1],P) :-  
    adjacent(X,Y,G),  
    \+ member(X,[Y | P1]),  
    path1(G,A,[X,Y | P1],P).
```

Graphs in Prolog

- Acyclic graph path:

`adjacent(X,Y,graph(_,Es)) :- member(e(X,Y),Es).`

`adjacent(X,Y,graph(_,Es)) :- member(e(Y,X),Es).`

Graphs in Prolog

- Cycle from a given node:
 - a predicate `cycle(G,A,P)` to find a closed path (cycle) `P` starting at a given node `A` in the graph `G`.
 - The predicate should return all cycles via backtracking.

```
cycle(G,A,P) :-  
    adjacent(B,A,G),  
    path(G,A,B,P1),  
    length(P1,L),  
    L > 2,  
    append(P1,[A],P).
```

Logical Puzzles

- Eight queens problem:
 - place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal.
 - We represent the positions of the queens as a list of numbers $1..N$ (e.g., $[4,2,7,3,6,8,5,1]$ means that the queen in the first column is in row 4, the queen in the second column is in row 2, etc.)

Logical Puzzles

- Eight queens problem:
 - place eight queens on a chessboard so that no two queens are attacking each other; i.e., no two queens are in the same row, the same column, or on the same diagonal.
 - We represent the positions of the queens as a list of numbers $1..N$ (e.g., $[4, 2, 7, 3, 6, 8, 5, 1]$ means that the queen in the first column is in row 4, the queen in the second column is in row 2, etc.)

Logical Puzzles

- Eight queens problem:
 - using the permutations of the numbers $1..N$ we guarantee that no two queens are in the same row
 - The only test that remains to be made is the diagonal test

Logical Puzzles

- Eight queens problem:

queens(N,Qs) :- range(1,N,Rs), perm(Rs,Qs), test(Qs).

% range(A,B,L) :- L is the list of numbers A..B

range(A,A,[A]).

range(A,B,[A | L]) :- A < B, A1 is A+1, range(A1,B,L).

% perm(Xs,Zs):-the list Zs is a permutation of the list Xs

perm([],[]).

perm(Qs,[Y | Ys]) :- del(Y,Qs,Rs), perm(Rs,Ys).

del(X,[X | Xs],Xs).

del(X,[Y | Ys],[Y | Zs]) :- del(X,Ys,Zs).

Logical Puzzles

- Eight queens problem:

`% test(Qs) :- Qs is a non-attacking queens solution`

`test(Qs) :- test(Qs,1,[],[]).`

`% test(Qs,X,Cs,Ds) :- the queens in Qs, representing
columns X to N, are not in conflict with the diagonals Cs
and Ds`

`test([],_,_,_).`

`test([Y | Ys],X,Cs,Ds) :-`

`C is X-Y, \+ memberchk(C,Cs),`

`D is X+Y, \+ memberchk(D,Ds),`

`X1 is X + 1, test(Ys,X1,[C | Cs],[D | Ds]).`

Aggregates in XSB

- `setof(?Template, +Goal, ?Set)` : ?Set is the set of all instances of Template such that Goal is provable.
- `bagof(?Template, +Goal, ?Bag)` has the same semantics as `setof/3` except that the third argument returns an unsorted list that may contain duplicates.
- `findall(?Template, +Goal, ?List)` is similar to predicate `bagof/3`, except that variables in Goal that do not occur in Template are treated as existential, and alternative lists are not returned for different bindings of such variables.
- `tfindall(?Template, +Goal, ?List)` is similar to predicate `findall/3`, but the Goal must be a call to a single tabled predicate.

XSB Prolog

- Negation: *not* ($\backslash +$): negation-as-failure
- Another negation called *tnot* (*TABLING* = *memoization*)
 - Use: ... :- ..., *tnot*(foobar(X)).
 - All variables under the scope of *tnot* must also occur to the left of that scope in the body of the rule in other positive relations:
 - Ok: ... :- p(X,Y), *tnot*(foobar(X,Y)), ...
 - Not ok: ... :- p(X,Z), *tnot*(foobar(X,**Y**)), ...
- XSB also supports Datalog:
 - :- auto_table.at the top of the program file

XSB Prolog

- Read/write from and to files:
 - Edinburgh style:
 - ?- see('a.txt'), read(X), seen.
 - ?- tell('a.txt'),
write('Hello, World!'), told.

XSB Prolog

- Read/write from and to files:
 - ISO style:
?- open('a.txt', write, X),
write(X, 'Hello, World!'),
close(X).

Cut (logic programming)

- Cut (! in Prolog) is a goal which always succeeds, **but cannot be backtracked past.**

- **Green cut**

`gamble(X) :- gotmoney(X), !.`

`gamble(X) :- gotcredit(X), \+ gotmoney(X).`

- **cut** says “stop looking for alternatives”
- by explicitly writing `\+ gotmoney(X)`, it guarantees that the second rule will always work even if the first one is removed by accident or changed

- **Red cut**

`gamble(X) :- gotmoney(X), !.`

`gamble(X) :- gotcredit(X).`

Cut (logic programming)

- Consider:

$p(a). p(b).$

$q(a). q(b). q(c).$

$?- p(X), !.$

$X=a ;$

no

$?- p(X), !, q(Y).$

$X=a Y=a ;$

$X=a Y=b ;$

$X=a Y=c ;$

Testing types

- **atom(X)**

Tests whether X is bound to a symbolic atom.

?- atom(a).

yes

?- atom(3).

no

- **integer(X)**

Tests whether X is bound to an integer.

- **real(X)**

Tests whether X is bound to a real number.

Testing for variables

- **ground(G)**

Tests whether G has unbound logical variables.

- **var(X)**

Tests whether X is bound to a Prolog variable.

Control / Meta-predicates

- **call(P)**

Force P to be a goal; succeed if P does, else fail.

Assert and retract

- **asserta(C)**

Assert clause C into database above other clauses with the same key predicate. The key predicate of a clause is the first predicate encountered when the clause is read from left to right.

- **assertz(C), assert(C)**

Assert clause C into database below other clauses with the same key predicate.

- **retract(C)**

Retract C from the database. C must be sufficiently instantiated to determine the predicate key.

Prolog terms and clauses

- **clause(H,B)**

Retrieves clauses in memory whose head matches H and body matches B. H must be sufficiently instantiated to determine the main predicate of the head.

- **functor(E,F,N)**

E must be bound to a functor expression of the form 'f(...)'. F will be bound to 'f', and N will be bound to the number of arguments that f has.

- **arg(N,E,A)**

E must be bound to a functor expression, N is a whole number, and A will be bound to the Nth argument of E

Prolog terms and clauses

- `=..`

converts between term and list. For example,

?- `parent(a,X) = .. L.`

`L = [parent, a, _X001]`

Definite clause grammar (DCG)

- A **DCG** is a way of expressing grammar in a logic programming language such as Prolog
- The definite clauses of a DCG can be considered a set of axioms where the fact that it has a parse tree can be considered theorems that follow from these axioms

DCG grammar for arithmetic expr.

expr --> term, addterm.

addterm --> [].

addterm --> [+], expr.

term --> factor, multifactor.

multifactor --> [].

multifactor --> [*], term.

factor --> [I], {integer(I)}.

factor --> ['('], expr, [')'].

% xsb

| ?- expr([4,*,5,+,1],[]).

yes

| ?- expr([1,+,3,*,'(',2,+,4,')'],[]).

yes

DCG grammar for arithmetic expr.

`:- table expr/3, term/3.`

`expr(Val) --> expr(Eval), [+], term(Tval), {Val is Eval+Tval}.`

`expr(Val) --> term(Val).`

`term(Val) --> term(Tval), [*], primary(Fval), {Val is Tval*Fval}.`

`term(Val) --> primary(Val).`

`primary(Val) --> ['(', expr(Val), ')'].`

`primary(Int) --> [Int], {integer(Int)}.`

`%xsb`

`| ?- [grammar].`

`| ?- expr(Val,[1,+,2,*,3,*,'(',4,+,5,')'],[]).`

`Val = 55`

A SIMPLE NATURAL LANGUAGE DCG

The cat scares the mouse.

| | | | |
det noun verb det noun

{
noun_phrase

{
noun_phrase

{
verb_phrase

{
sentence

A SIMPLE NATURAL LANGUAGE DCG

sentence --> noun_phrase, verb_phrase.

verb_phrase --> verb, noun_phrase.

noun_phrase --> determiner, noun.

determiner --> [the].

noun --> [cat].

noun --> [cats].

noun --> [mouse].

verb --> [scares].

verb --> [scare].

?- sentence(X,[]).

Context-free grammar

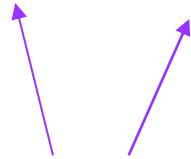
THIS GRAMMAR GENERATES

[the, cat, scares, the, mouse]

[the, mouse, scares, the, mouse]

[the, cats, scare, the, mouse]

[the, cats, scares, the, mouse]



CONTEXT DEPENDENT!

NUMBER AGREEMENT CAN BE FORCED BY ARGUMENTS

sentence(**Number**) -->

 noun_phrase(**Number**), verb_phrase(Number).

verb_phrase(**Number**) -->

 verb(**Number**), noun_phrase(Number1).

noun_phrase(Number) -->

 determiner(Number), noun(Number).

noun(singular) --> [mouse].

noun(plural) --> [mice].

verb(singular) --> [scares].

verb(plural) --> [scare].

?- sentence(X, Number, []).

Context-sensitive grammar

DCG with Parse tree

`sentence(s(NP,VP)) --> noun_phrase(NP), verb_phrase(VP).`

`noun_phrase(np(D,N)) --> det(D), noun(N).`

`verb_phrase(vp(V,NP)) --> verb(V), noun_phrase(NP).`

`det(d(the)) --> [the].`

`det(d(a)) --> [a].`

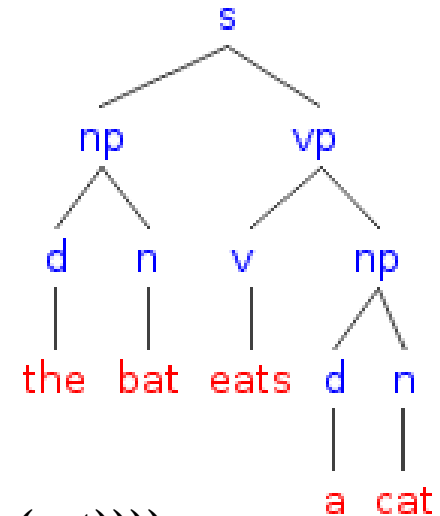
`noun(n(bat)) --> [bat].`

`noun(n(cat)) --> [cat].`

`verb(v(eats)) --> [eats].`

`?- sentence(Parse_tree, [the,bat,eats,a,cat], []).`

`Parse_tree = s(np(d(the),n(bat)),vp(v(eats),np(d(a),n(cat))))`



DCG is just syntactic sugar in Prolog

% special rule syntax

s --> np, vp.

np --> det, n.

vp --> tv, np.

vp --> v.

det --> [the].

det --> [a].

det --> [every].

n --> [man].

n --> [woman].

n --> [park].

tv --> [loves].

tv --> [likes].

v --> [walks].

% Grammar in pure Prolog

s(S0,S) :- np(S0,S1), vp(S1,S).

np(S0,S) :- det(S0,S1), n(S1,S).

vp(S0,S) :- tv(S0,S1), np(S1,S).

vp(S0,S) :- v(S0,S).

det(S0,S) :- S0=[the|S].

det(S0,S) :- S0=[a|S].

det(S0,S) :- S0=[every|S].

n(S0,S) :- S0=[man|S].

n(S0,S) :- S0=[woman|S].

n(S0,S) :- S0=[park|S].

tv(S0,S) :- S0=[loves|S].

tv(S0,S) :- S0=[likes|S].

v(S0,S) :- S0=[walks|S].

?- s([a,man,loves,the,woman],[]).

yes

Prolog Programming

- **Recursion is king to Computational Problem Solving**
 - learn to express algorithmic ideas in an abstract manner
- Prolog Programming Contest:
 - <http://people.cs.kuleuven.be/~bart.demoen/PrologProgrammingContests/>
 - First 10 contests book:
<https://dtai.cs.kuleuven.be/ppcbook/ppcbook.pdf>
 - Fun fact: the winning Stony Brook team in 1998
 - <http://people.cs.kuleuven.be/~bart.demoen/PrologProgrammingContests/contest98.html>



Prolog Programming

- Patterns: Triangle (1995 Portland, USA)
 - Write a predicate `triangle/1`, which is called with its argument `N` instantiated to a non-negative integer, and which draws a triangle of size `N` on the screen:

The diagram illustrates the recursive construction of a triangle pattern of size 7. It shows the pattern for size 7 on the left, followed by an equals sign, then the pattern for size 6 followed by a plus sign and the pattern for size 1. The patterns are composed of asterisks (*) and underscores (_).

Size 7 pattern:

```
_ _ _ _ _ _ *  
_ _ _ _ _ * *  
_ _ _ _ * * *  
_ _ _ * * * *  
_ _ * * * * *  
_ * * * * * *  
_
```

Size 6 pattern:

```
_ _ _ _ _ _  
_ _ _ _ _  
_ _ _ _  
_ _ _  
_ _  
_
```

Size 1 pattern:

```
*
```

Prolog Programming

- Patterns: Triangle (1995 Portland, USA)

```
triangle(N) :- Stars = 1,
```

```
    triangle(N, Stars) .
```

```
triangle(_).
```

```
triangle(Spaces, Stars) :-
```

```
    Spaces > 0, writeN(Spaces, ' '),
```

```
    writeN(Stars, '* '), nl,
```

```
    Spaces1 is Spaces - 1,
```

```
    Stars1 is Stars + 1,
```

```
    triangle(Spaces1, Stars1) .
```


The future of languages

- That is all!
- My guess on where languages will be going: languages that combine:
 - Multiparadigm,
 - High-level data structures,
 - With: speed, simplicity (dynamic weakly typed).
- JavaScript frameworks, node.js, Google Go, Swift, and what else?
 - More and more languages every day!!! What should we learn? All!
 - Youtube is implemented with Python,
 - IBM Watson uses Prolog,
 - Wikipedia is implemented with PHP,
 - Microsoft F# is a functional programming language, etc.
 - More Scripting Languages: Writing programs by coordinating pre-existing components, rather than writing components from scratch.

The future of languages

- Scripting-style:
 - Speed : Trade almost everything for developer productivity
 - Economy of Expression
 - Lack of Declarations
 - Simple rules
 - Flexible dynamic typing
 - Access to the OS
 - Sophisticated string processing
 - High-level data structures: Maps, Lists, Tuples, Sets.
 - Batch and Interactive
 - Open and Portable
 - Single Canonical Implementation
 - Interpreted: Fast to start
 - Easily extended
 - Easily embedded

The future of languages

- I'm hoping that this course prepared you for the change the future will bring
- Thank you!