Case Study
Logic Programming

Outline

- Preliminary Concepts
- Horn Clauses
- Logic Programming in Prolog
- Prolog Program Elements
- Practical Aspects of Prolog
- Prolog Examples
  - Solving Word Puzzles
  - Natural Language Processing

Computation vs. Deduction

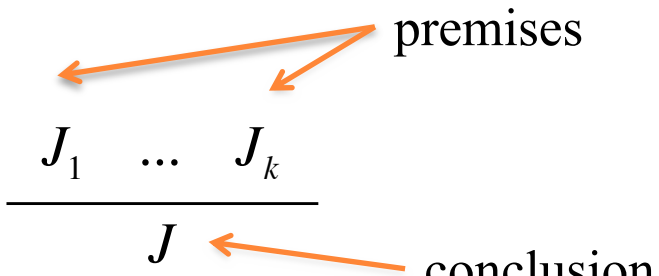
- Compute:
  - An expression + rules (operational semantics) → result
- Deduce:
  - A conjecture + rules (axioms/inference rules) → a proof
- Logic Programming unites these two:
  - If we fix a strategy for proof search, then deduction can be considered a computation
  - Strategy == algorithm in logic

Proof System

- Recall Curry-Howard Isomorphism:

  Logic == Type system

- Inference rule:



- E.g.,

$$\frac{}{even(z)}\ \text{(even-z)} \qquad \frac{even(N)}{even(s(s(N)))}\ \text{(even-s)}$$

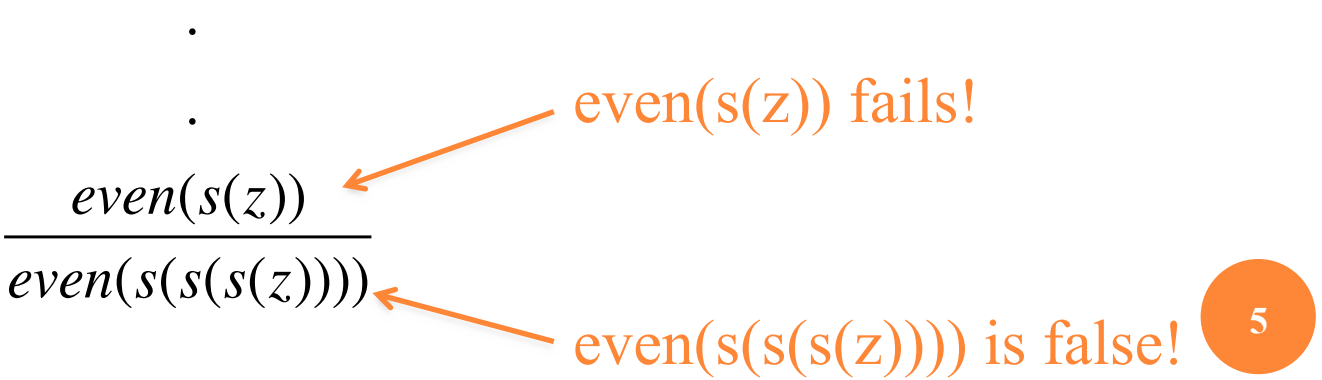
- Deduction:

$$\dfrac{\dfrac{\dfrac{}{even(z)}}{even(s(s(z)))}}{even(s(s(s(s(z)))))} \quad \begin{array}{l}\text{(even-z)}\\ \text{(even-s)}\\ \text{(even-s)}\end{array}$$

Proof Search

- Search for the right rule to apply at each step.
- Two strategies:
  - Backward reasoning (goal-directed): from conjecture (a.k.a. *goal*) to axioms.
  - Forward reasoning: from axioms to conjecture.

- *Negation as failure*



Answer Substitution

- Previous example tells us if a number is even or not (yes if there is a proof, and no otherwise).
- LP also compute values
- Define natural number addition:

$$\frac{}{add(z,N,N)}\ \text{(add-z)} \qquad \frac{add(N,M,P)}{add(s(N),M,s(P))}\ \text{(add-s)}$$

- Consider the conjecture (goal): *add (s(z), s(z), R).*
  - Not only prove that this goal is true.
  - Also want to know the value of R.
  - Search not only constructs a proof, but also searches for a value R that makes the goal hold.

Answer Substitution

- Do the deduction on two rules:

$$\dfrac{\dfrac{add(z,s(z),s(z))}{add(z,s(z),P)}}{add(s(z),s(z),R)}$$

(add-z) and $P=s(z)$

(add-s) and $R=s(P)$

- Substitute $s(z)$ for P, we get $R = s(s(z))$, i.e.,

$$\dfrac{\dfrac{add(z,s(z),s(z))}{add(z,s(z),s(z))}}{add(s(z),s(z),s(s(z)))}$$

Backtracking

- When a goal matches the conclusion of more than one rule: we reach a *choice point*.
- At the choice point, we pick a rule and attempt the proof.
- If that attempt fails, we go back to the most recently choice point, pick another rule.
- This process is called *backtracking*.

Backtracking

- Consider goal: *add(M, s(z), s(s(z)))*
  - Computing M = 2 – 1

$$\frac{add(M_1, s(z), s(z))}{add(M, s(z), s(s(z))} \quad \text{(add-s) and } M = s(M_1)$$

- Both rule (add-s) and rule (add-z) can fire.
- If we pick (add-s):

Fail! →

$$\frac{\dfrac{add(M_2, s(z), z)}{add(M_1, s(z), s(z))}}{add(M, s(z), s(s(z))} \quad \begin{matrix} \text{(add-s) and } M_1 = s(M_2) \\ \text{(add-s) and } M = s(M_1) \end{matrix}$$

Backtracking

- Backtrack and try (add-z):

Succeed! →

$$\frac{}{add(M_1, s(z), s(z))}$$

$$add(M, s(z), s(s(z)))$$

(add-z) and $M_1 = z$

(add-s) and $M = s(M_1)$

Subgoal Order

- When a rule contains multiple premises, we need to determine which premise (or subgoal) to attempt first.

$$\frac{J_1 \quad \ldots \quad J_k}{J}$$

- The order of subgoal evaluation has a significant impact on the computation:
  - Complete in a few steps in some order
  - Non-terminating in other order
  - Corresponds to order of search (DFS, BSF, etc.)

Horn Clauses

- An inference rule can be represented as a *horn clause*.
- A Horn clause has a head h, which is a *predicate*, and a body, which is a list of *predicates* $p_1, p_2, \ldots, p_n$.
- It is written as:

$$h \leftarrow p_1, p_2, \ldots, p_n$$

- This means, "h is true only if $p_1, p_2, \ldots,$ and p_n are simultaneously true."
- E.g., the Horn clause:

$$snowing(C) \leftarrow precipitation(C), freezing(C)$$

says, "it is snowing in city C only if there is precipitation in city C and it is freezing in city C."

Horn Clauses and Predicates

- Any Horn clause

$$h \leftarrow p_1, p_2, \ldots, p_n$$

can be written as a predicate:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \supset h$$

or equivalently:

$$\neg(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \vee h$$

- But not every predicate can be written as a Horn clause, such as *disjunctions:*
- E.g., $literate(x) \supset reads(x) \vee writes(x)$

Resolution and Unification

- Resolution:

  If h is the head of a Horn clause

  $$h \leftarrow terms$$

  and it matches one of the terms of another Horn clause:

  $$t \leftarrow t_1, h, t_2$$

  then that term can be replaced by h's terms to form:

  $$t \leftarrow t_1, terms, t_2$$

- During resolution, assignment of variables to values is called *instantiation*.

- *Unification* is a pattern-matching process that determines what particular instantiations can be made to variables during a series of resolutions. (Similar to the unification machine in type inference)

Example

- The two clauses:

  speaks(*Mary*, *English*)

  talkswith(*X*, *Y*) ← *speaks*(*X*, *L*), *speaks*(*Y*, *L*), *X*≠*Y*

can resolve to:

talkswith(*Mary*, Y) ←

speaks(*Mary*, *English*),

speaks(*Y*, *English*), Mary≠*Y*

- The assignment of values *Mary* and *English* to the variables *X* and *L* is an *instantiation* for which this *resolution* can be made.

Logic Programming in Prolog

- In logic programming the program declares the goals of the computation, not the method for achieving them.

- Logic programming has applications in AI and databases.
  - Natural language processing (NLP)
  - Automated reasoning and theorem proving
  - Expert systems (e.g., MYCIN)
  - Database searching, as in SQL (Structured Query Language)

- Prolog emerged in the 1970s. Distinguishing features:
  - Nondeterminism
  - Backtracking

Prolog Program Elements

- Prolog programs are made from *terms*, which can be:
  - Variables
  - Constants
  - Structures
- *Variables* begin with a capital letter, like Bob.
- *Constants* are either integers, like 24, or atoms, like the, zebra, 'Bob', and '.'.
- *Structures* are predicates with arguments, like:

  n(zebra), speaks(Y, English), and np(X, Y)
- The arity of a structure is its number of arguments (1, 2, and 2 for these examples).

Facts, Rules, and Programs

- A Prolog *fact* is a Horn clause without a right-hand side. Its form is (note the required period .):

  term.

- A Prolog *rule* is a Horn clause with a right-hand side. Its form is (note :- represents ← and a period . Is required):

  term :- $term_1$, $term_2$, … $term_n$.

- A Prolog *program* is a collection of facts and rules.

Example Program

```
speaks(allen, russian).
speaks(bob, english).
speaks(mary, russian).
speaks(mary, english).
talkswith(X, Y) :- speaks(X, L), speaks(Y, L), X \= Y.
```

- This program has four facts and one rule.
- The rule *succeeds* for any instantiation of its variables in which all the terms on the right of `:-` are simultaneously true. E.g., this rule succeeds for the instantiation `X=allen`, `Y=mary`, and `L=russian`.
- For other instantiations, like `X=allen` and `Y=bob`, the rule *fails*.

Searching for Success: Queries

- A *query* is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interest. E.g.,

```
?- speaks(Who, russian).
```

asks for an instantiation of the variable Who for which the query speaks(Who, russian) succeeds.

- A program is loaded by the query consult, whose argument names the program. E.g.,

```
?- consult(speaks).
```

loads the program named speaks, given on the previous slide.

- The word "consult" is reminiscent of the "expert system".

Answering the Query: Unification

- To answer the query:

  ?- speaks(Who, russian).

- Prolog considers every fact and rule whose head is speaks. (If more than one, consider them in *some* order – non-deterministic!)

- Resolution and unification locate all the successes:

  Who = allen ;

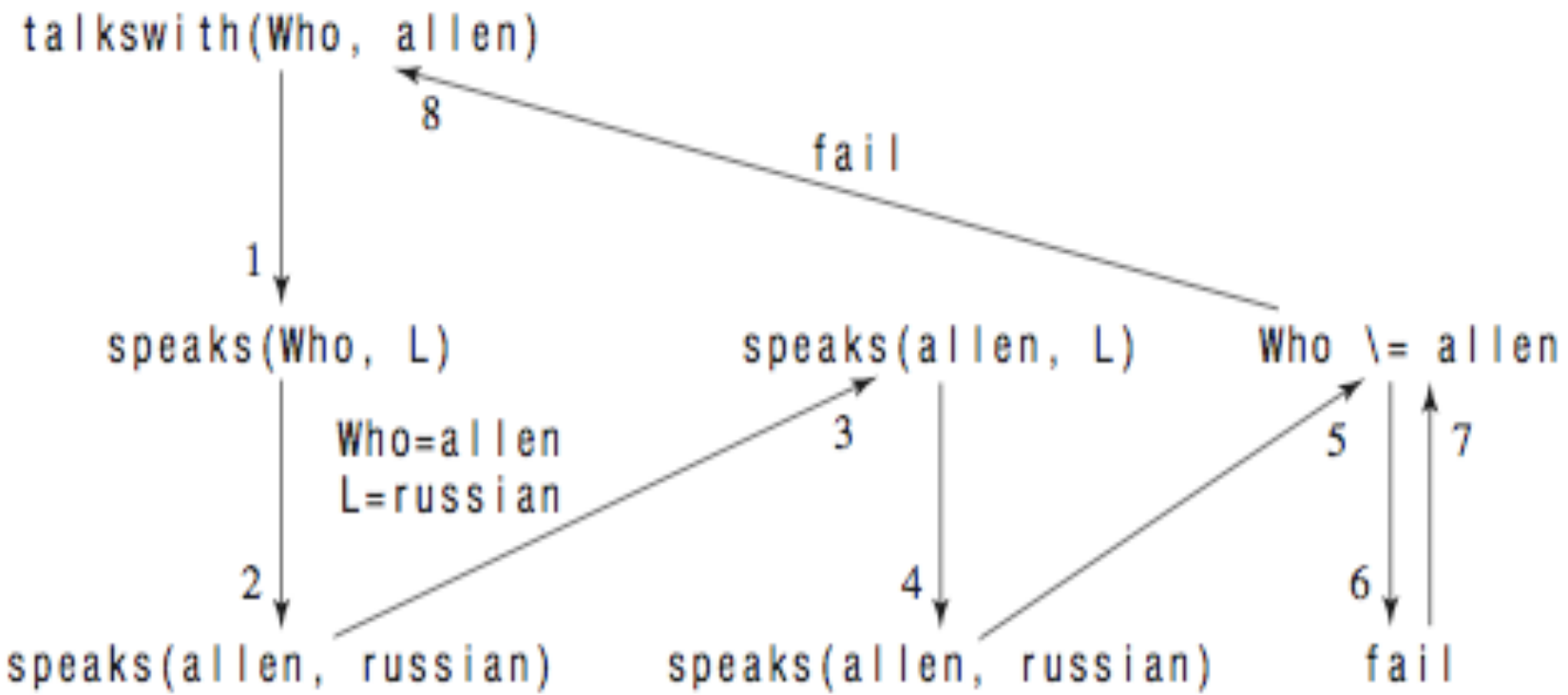
  Who = mary ;

  No

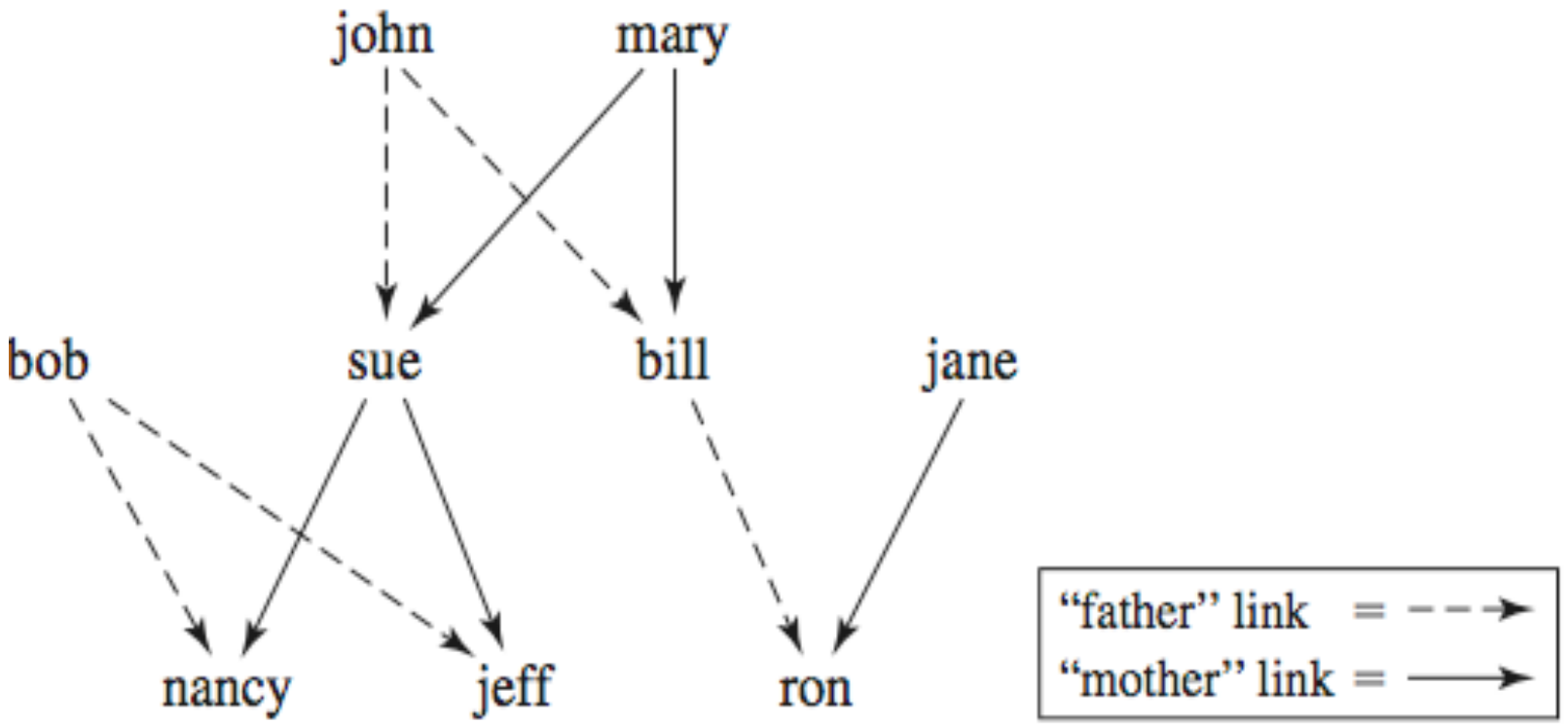
  - Each semicolon (;) asks, "Show me the next success."

Search Trees

- First attempt to satisfy the query ?- talkswith(Who, allen).



Database Search - The Family Tree



Prolog Program

```
mother(mary, sue).
mother(mary, bill).
mother(sue, nancy).
mother(sue, jeff).
mother(jane, ron).

father(john, sue).
father(john, bill).
father(bob, nancy).
father(bob, jeff).
father(bill, ron).

parent(A,B) :- father(A,B).
parent(A,B) :- mother(A,B).
grandparent(C,D) :- parent(C,E), parent(E,D).
```

Some Database Queries

- Who are the parents of jeff?

```
?- parent(Who, jeff).
Who = bob;
Who = sue
```

- Find all the grandparents of Ron.

```
?- grandparent(Who, ron).
```

- What about siblings? Those are the pairs who have the same parents.

```
?- sibling(X, Y) :- parent(W, X), parent(W, Y), X\=Y.
```

Lists

- A *list* is a series of terms separated by commas and enclosed in brackets.
  - The empty list is written [].
  - The sentence "The giraffe dreams" can be written as a list:
    [the, giraffe, dreams]
  - A "don't care" entry is signified by _, as in
    [_, X, Y]
  - A list can also be written in the form:
    [Head | Tail]
  - The functions
    append joins two lists, and
    member tests for list membership.

APPEND FUNCTION

```
append([], X, X).
append([Head | Tail], Y, [Head | Z]) :-
                                    append(Tail, Y, Z).
```

- This definition says:

  1. Appending any list (X) to the empty list returns an unchanged list (X again).

  2. If Y is appended to Tail to get Z, then Y can be appended to a list one element larger [Head | Tail] to get [Head | Z].

- *Note*: The last parameter designates the result of the function. So a variable must be passed as an argument.

MEMBER FUNCTION

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

- The test for membership succeeds if either:
  1. X is the head of the list [X | _]
  2. X is not the head of the list [_ | Y] , but X is a member of the list Y.

- Notes: *pattern matching* governs tests for equality.
- Don't care entries (_) mark parts of a list that aren't important to the rule.

More List Functions

- X is a *prefix* of Z if there is a list Y that can be appended to X to make Z. That is:

```
prefix(X, Z) :- append(X, Y, Z).
```

- Similarly, Y is a *suffix* of Z if there is a list X to which Y can be appended to make Z. That is:

```
suffix(Y, Z) :- append(X, Y, Z).
```

- So finding all the prefixes (suffixes) of a list is easy. E.g.:

```
?- prefix(X, [my, dog, has, fleas]).
X = [];
X = [my];
X = [my, dog];
...
```

Practical Aspects of Prolog

- Tracing
- The Cut
- Negation
- The is, not, and Other Operators
- The Assert Function

TRACING

- To see the dynamics of a function call, the trace function can be used. E.g., if we want to trace a call to the following function:

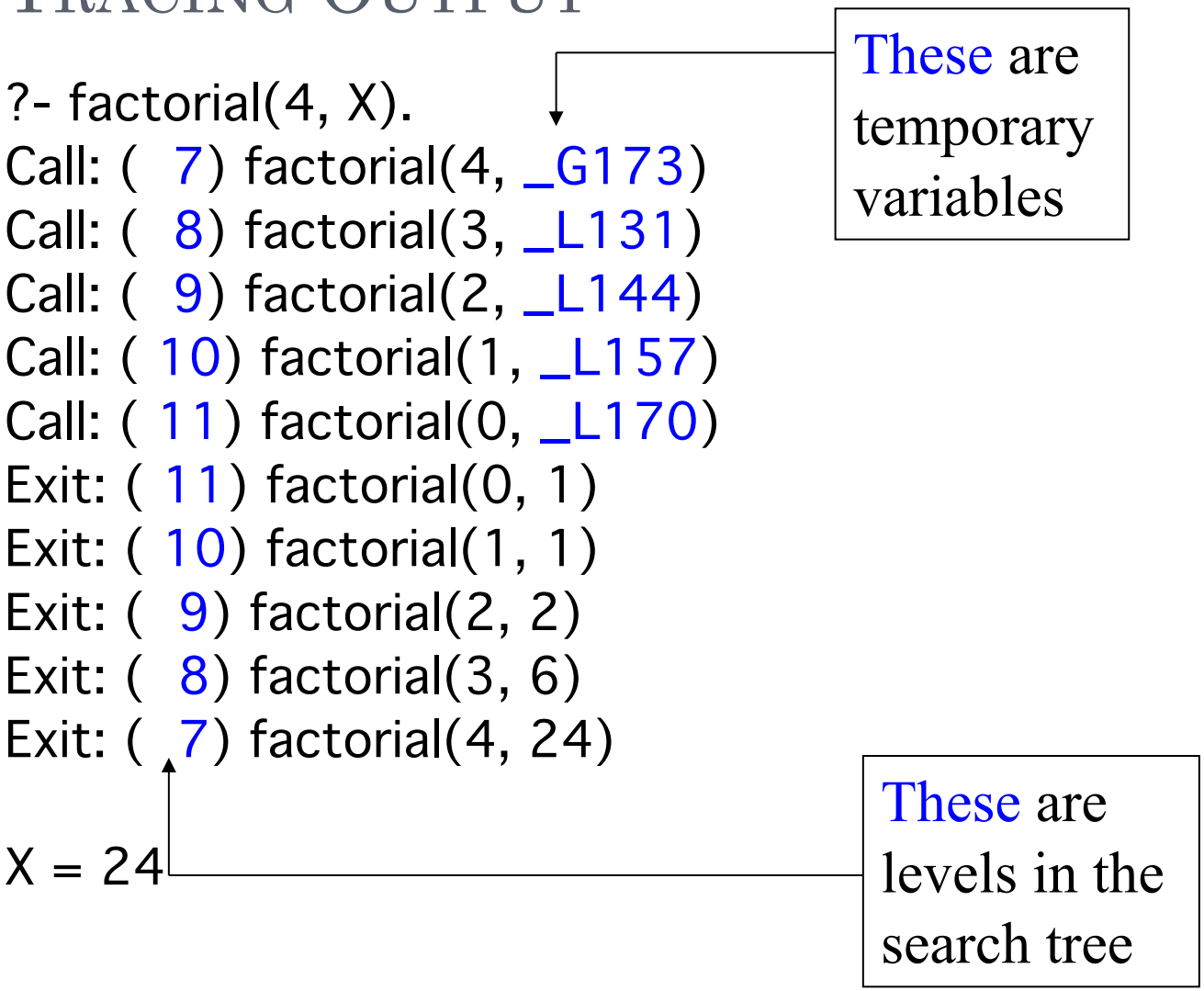
```
factorial(0, 1).
factorial(N, Result) :- N > 0, M is N - 1,
        factorial(M, SubRes), Result is N * SubRes.
```

- we can activate trace and then call the function:

```
?- trace(factorial/2).
?- factorial(4, X).
```

- *Note*: the argument to trace must include the function's arity.

Tracing Output



The Cut

- The *cut* is an operator (!) inserted on the right-hand side of a rule.
- *semantics*: the cut forces the subgoals to its left not to be retried if the right-hand side succeeds once, i.e. no backtrack to the left of (!).
- E.g (bubble sort):

```
bsort(L, S) :- append(U, [A, B | V], L),
               B < A, !,
               append(U, [B, A | V], M),
               bsort(M, S).
bsort(L, L).
```

- So this code gives one answer rather than many.
- Limit search space and improves performance.

Bubble Sort Trace

```
?- bsort([5,2,3,1], Ans).
Call:  (  7) bsort([5, 2, 3, 1], _G221)
Call:  (  8) bsort([2, 5, 3, 1], _G221)
...
Call:  ( 12) bsort([1, 2, 3, 5], _G221)
Redo:  ( 12) bsort([1, 2, 3, 5], _G221)
...
Exit:  (  7) bsort([5, 2, 3, 1], [1, 2, 3, 5])

Ans = [1, 2, 3, 5] ;

No
```

Without the cut, this would have given some wrong answers.

The *is* Operator

- is instantiates a temporary variable. E.g., in

```
factorial(0, 1).
factorial(N, Result) :- N > 0, M is N - 1,
                        factorial(M, SubRes), Result is N * SubRes.
```

- Here, the variables M and Result are instantiated This is like an assignment to a local variable in C-like languages.
- You could have said:

```
factorial(N, N * SubRes) :- N > 0,  factorial(N - 1, SubRes).
```

Other Operators

- Prolog provides the operators

  `+ - * / ^ = < > >= =< \=`

with their usual interpretations.

- The not operator is implemented as goal failure. E.g.,

```
factorial(N, 1) :- N < 1.
factorial(N, Result) :- not(N < 1), M is N - 1,
                        factorial(M, P),
                        Result is N * P.
```

is equivalent to using the cut (N < 1!) in the first rule.

The assert Function

- The assert function can update the facts and rules of a program dynamically. E.g., if we add the following to the foregoing database program:

```
?- assert(mother(jane, joe)).
```

- Then the query:

```
?- mother(jane, X).
```

- gives:

```
X = ron ;
X = joe;
No
```

Prolog Examples

1. Solving Word Puzzles

   Nondeterminism seeks all solutions, not just one

2. Natural Language Processing

   One of Prolog's traditional research applications

Solving Word Puzzles

- A simple example:

  Baker, Cooper, Fletcher, Miller, and Smith live in a five-story building. Baker doesn't live on the 5th floor and Cooper doesn't live on the first. Fletcher doesn't live on the top or the bottom floor, and he is not on a floor adjacent to Smith or Cooper. Miller lives on some floor above Cooper. Who lives on what floors?

- We can set up the solution as a list of five entries:

```
[floor(_, 5), floor(_, 4), floor(_, 3), floor(_, 2), floor(_, 1)]
```

- The *don't care* entries are placeholders for the five names.

Modeling the solution

- We can identify the variables B, C, F, M, and S with the five persons, and the structure floors(Floors) as a function whose argument is the list to be solved.

- Here's the first constraint:

```
member(floor(baker,  B), Floors), B\=5
```

which says that *Baker doesn't live on the 5th floor*.

- The other four constraints are coded similarly, leading to the following program:

Prolog solution

```
floors([floor(_,5),floor(_,4),floor(_,3),floor(_,2),
                                        floor(_,1)]).
building(Floors) :- floors(Floors),
    member(floor(baker, B), Floors), B \= 5,
    member(floor(cooper, C), Floors), C \= 1,
    member(floor(fletcher, F), Floors), F \= 1, F \= 5,
    member(floor(miller, M), Floors), M > C,
    member(floor(smith, S), Floors), not(adjacent(S, F)),
    not(adjacent(F, C)),
    print_floors(Floors).
```

Auxiliary functions

- Floor adjacency:

```
adjacent(X, Y) :- X =:= Y+1.
adjacent(X, Y) :- X =:= Y-1.
```

Note: =:= tests for numerical equality.

Displaying the results:

```
print_floors([A | B]) :- write(A),  nl,
print_floors(B).
print_floors([]).
```

Note: write is a Prolog function and nl stands for "new line."

- Solving the puzzle is done with the query:

```
?- building(X).
```

which finds an instantiation for X that satisfies all the constraints.

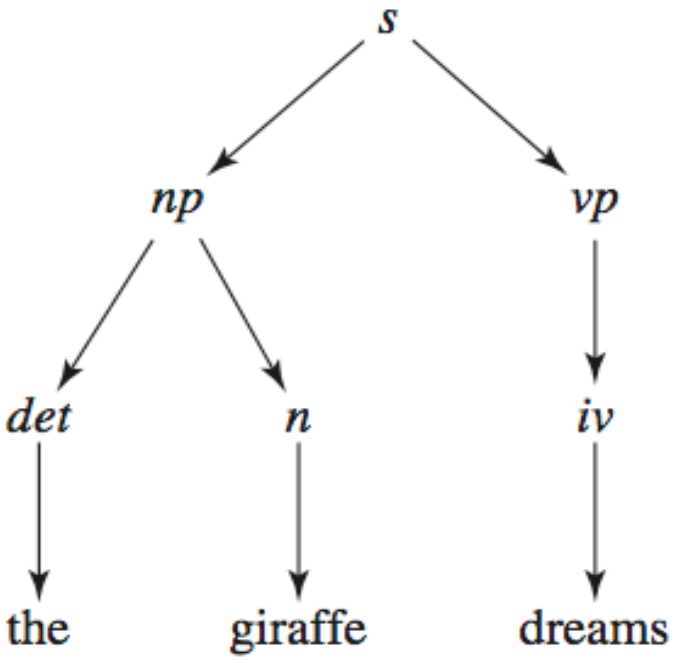
Natural Language Processing

- BNF can define natural language (e.g., English) syntax.
  - This was the original purpose of BNF when it was invented by Chomsky in 1957.
- A Prolog program can model a BNF grammar.
  - This was an original purpose of Prolog when it was designed in the 1970s.
- A Prolog list can model a sentence.
  - E.g., [the, giraffe, dreams]
- So running the program can parse a sentence.

NLP Example

Consider the following BNF grammar and parse tree:

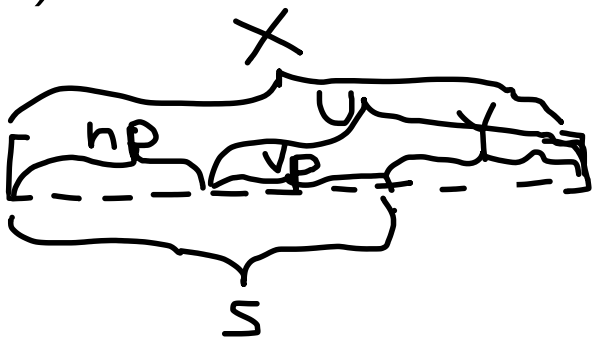
$s \rightarrow np\ vp$
$np \rightarrow det\ n$
$vp \rightarrow tv\ np$
$\rightarrow iv$
$det \rightarrow$ the
$n \rightarrow$ giraffe
$\rightarrow$ apple
$iv \rightarrow$ dreams
$tv \rightarrow$ eats



Here, *s*, *np*, *vp*, *det*, *n* , *iv*, and *tv* denote "sentence," "noun phrase," "verb phrase," "determiner," "noun," "intransitive verb," and "transitive verb."

Prolog Encoding (naïve version)

```
s(X, Y) :- np(X, U), vp(U, Y).
np(X, Y) :- det(X, U), n(U, Y).
vp(X, Y) :- iv(X, Y).
vp(X, Y) :- tv(X, U), np(U, Y).
det([the | Y], Y).
n([giraffe | Y], Y).
n([apple | Y], Y).
iv([dreams | Y], Y).
tv([eats | Y], Y).
```



- The first rule reads, "list X is (a sentence plus a tail Y) if X is (a noun phrase plus a tail U) and U is (a verb phrase plus a tail Y)."

Example Trace

The query asks, "can you resolve [the, giraffe, dreams] as an s, leaving tail []?"

```
?- s([the, giraffe, dreams],[]).
Call:  (  7) s([the, giraffe dreams], []) ?
Call:  (  8) np([the, giraffe, dreams], _L131) ?
Call:  (  9) det([the, giraffe, dreams], _L143) ?
Exit:  (  9) det([the, giraffe, dreams], [giraffe, dreams]) ?
Call:  (  9) n([giraffe, dreams], _L131) ?
Exit:  (  9) n([giraffe, dreams], [dreams]) ?
Exit:  (  8) np([the, giraffe, dreams], [dreams]) ?
Call:  (  8) vp([dreams], []) ?
Call:  (  9) iv([dreams], []) ?
Exit:  (  9) iv([dreams], []) ?
Exit:  (  8) vp([dreams], []) ?
Exit:  (  7) s([the, giraffe, dreams], []) ?
Yes
```

The result is success.

Definite Clause Grammars (DCGs)

```
s --> np, vp.
np --> det, n.
vp --> iv.
vp --> tv, np.
det --> [the].
n --> [giraffe].
n --> [apple].
iv --> [dreams].
tv --> [eats].
```

This replaces

```
s(X, Y) :- np(X, U), vp(U, Y).
```

but it means the same thing.

- *Note*: This form looks more like a series of BNF rules, and it's simple to write!

Generating Parse Trees

- Terms and variables can be added to the DCG rules so that a parse tree can be generated from a query.
- E.g., if we change

```
s --> np, vp.
```

to

```
s(s(NP, VP)) --> np(NP), vp(VP).
```

- The variables NP and VP can capture intermediate subtrees.

- When all rules are augmented in this way, the query

```
?- s(Tree, [the, giraffe, dreams], []).
```

delivers the parse tree as a parenthesized list:

```
Tree = s(np(det(the), n(giraffe)), vp(iv(dreams)))
```