Prolog

Computers Playing Jeopardy! Course

Stony Brook University

- IBM Watson Question Analysis for Jeopardy! = UIMA + Prolog + NLP

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.

What Is Prolog?

- Prolog is a logic-based language
- Simple Knowledge Representation
- With a few simple rules, information can be analyzed
  - Socrates is a man.
  - All men are mortal.
    - Therefore, Socrates is mortal.
- This is logic. Can Prolog do it?
  - Yes, but infinite inference in some cases
- XSB = Prolog + tabling
  - better termination properties

Brief History

- The first, official version of Prolog was developed
  - at the University of Marseilles, France by Alain Colmerauer in the early 1970s
  - as a tool for PROgramming in LOGic.
- Fifth generation project in Japan.
  - Europe and US followed.
- Many rule systems today in: natural language processing, semantic Web, access control system, expert systems, etc.
- Several Prolog systems in use: XSB Prolog, SWI, Yap, GNU Prolog, Sicstus, Quitus, Ciao, EcliPse, etc.
  - Extensions: Flora-2 F-logic, Answer Set Programming, OntoBroker.

Application Areas

- Prolog has been a very important tool in:
  - artificial intelligence applications
  - expert systems
  - natural language processing
  - smart information management systems
  - business rules
  - security access control policies

Declarative Language

- Prolog is based on mathematical logic with deductive capabilities.
- This means that
  - The programmer
    - declares **facts** (true facts)
    - defines **rules** (logical implication) for reasoning with the facts
  - Prolog uses deductive reasoning to
    - decide whether a proposed fact (**goal, query**) can be logically derived from known facts (such a decision is called a **conclusion**)
    - determine new facts from old

Prolog: Facts, Rules and Queries

	Prolog
Socrates is a man.	*man(socrates).*
All men are mortal.	*mortal(X) :- man(X).*
Is Socrates mortal?	*?- mortal(socrates).*
	Yes

Formalizing Arguments

- Abstracting with symbols for predicates, we get an argument form that looks like this:

  if p then q
  p is true
  therefore q is true

$((q :- p) \wedge p) \Rightarrow q$

Declarative Language Monotonic logic

- Standard logic is monotonic: once you prove something is true, it is true forever.
- Logic isn't a good fit to reality (we don't know everything and we learn true things that we believed in the past to be False).
- Prolog's *not* (\+) operator is a *closed-world* ***negation as failure:*** if no proof can be found for the fact, then the negative goal succeeds.
  - Example: *illegal(X) :- \+ legal(X).*
    - Example inference: *If we don't know anything legal about "Al Capone", then it must be the case that he is doing something illegal is true.*
  - This is non-monotonic:
    - Adding a fact that something is legal destroys an argument that it is illegal.

Declarative Language Non-monotonic logic

- A non-monotonic logic is a formal logic whose consequence relation is not monotonic.
  - Adding a formula to a theory produces a reduction of its set of consequences.

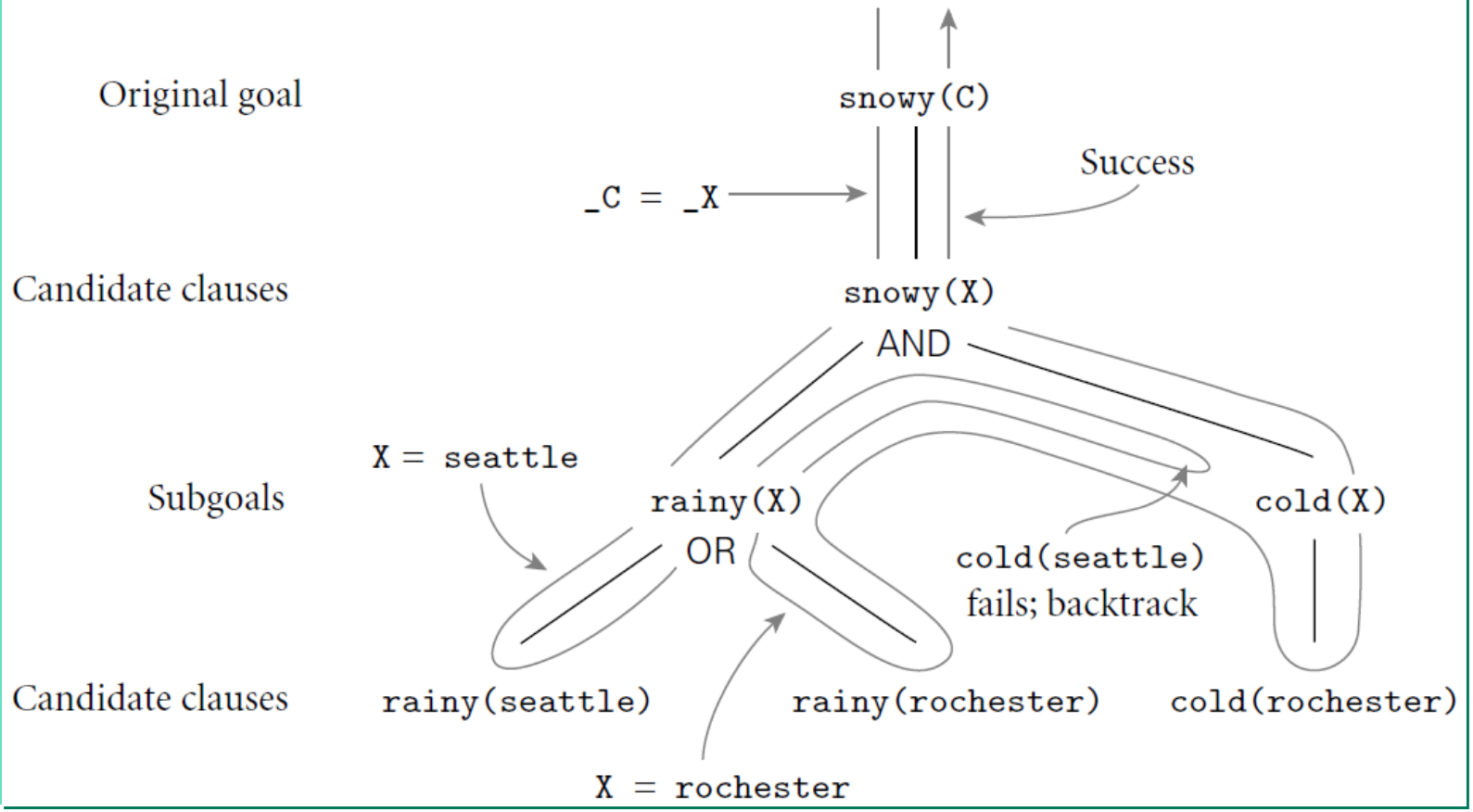
    p:- \+ *q*.
  - *p* is true because *q* is not **known/derivable** to be true.
  - What if later *q* is **asserted**? Then *p* is false (it destroyed the proof that it is true).
  - The \+/1 prefix operator is also called the "not provable" operator, since the query ?- \+ *Goal*. succeeds if *Goal* is not provable.

Forward and backward reasoning

- A syllogism gives two premises.
  - We ask: "**What is everything that we can conclude?**"
  - This is **forward reasoning** -- from premises to conclusions!
  - it's inefficient when you have lots of premises: many things can be inferred.
- Instead, you ask Prolog specific questions:
  - Example: “Can we infer that Al is a criminal?”
  - Prolog seeks for the goals provided by the user as questions
  - Prolog uses backward reasoning -- from (potential) conclusions to facts
    - Prolog searches successful paths and if it reaches unsuccessful branch, it backtracks to previous one and tries to apply alternative clauses

Prolog Backtracking

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



Logic Programming Concepts

- Based on predicate calculus
- Predicates: $p(a_1, a_2, \ldots, a_n)$
  - Example: a graph declared with facts (true statements)

    edge(a, b).

    edge(a, c).

    edge(b, d).

    edge(c, d).
  - Rules:

    1) if there's an edge from X to Y, we can reach Y from X.

    reach(X, Y) :- edge(X, Y).
    - ":-" means "implied by"

    2) if there's an edge from X to Y, and we can reach Z from Y, we can reach Z from X.

    reach(X,Z) :- edge(X, Y), reach(Y, Z).
    - "," means *and* (conjunction), ";" means *or* (disjunction).

Running XSB Prolog

- Install XSB Prolog
  - Windows distribution
  - build/configure and make for Linux and MacOS
- Create your "database" (program) in any editor

  man(socrates).

  mortal(X) :- man(X).
- Save it as *text only*, with a **.P** extension (or .pl)
- Run xsb

  ?- consult('socrates.pl').
- Then, ask your question at the prompt:

  ?- mortal(socrates).

Some Useful Tricks

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

```
| ?- q(X).    <Return>
X = 2    ;  <Return>
X = 4       <Return>
yes
| ?-
```

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

```
| ?- (q(_X), write('X='), write(_X),nl, 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 _.)

Prolog is a theorem prover

- Prolog's "Yes" means "I can prove it“
- Prolog's "No" means "I can't prove it"

  ?- mortal(plato).
  No

  - XSB Prolog has closed world assumption: knows everything it needs to know at a given time, so “**No**” means that the fact cannot be proven with the current information (and negation of failure).
- Prolog supplies values for variables when it can
  - ?- mortal(X).
    X = socrates

Prolog Example: Reachability

```
edge(1,2).
edge(2,3).
edge(2,4).
reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z), reachable(Z, Y).
```

Prolog Example: Reachability

```
 | ?- reachable(X,Y).

X = 1
Y = 2;    Type a semi-colon repeatedly

X = 2
Y = 3;

X = 2
Y = 4;

X = 1
Y = 3;

X = 1
Y = 4;

no


| ?- halt.    Command to Exit XSB
```

XSB Prolog Example: Reachability

Cycles in the graph may generate infinite loops

- **We can table (remember) queries and results, so we don't ask the same queries over and over again.**

```
edge(1,2).
edge(2,3).
edge(2,4).
edge(4,1).
:- table(reachable/2).
reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z), reachable(Z, Y).
```

Prolog in detail

- A *predicate* is a collection of clauses with the same *functor* (name) and *arity* (number of arguments).

```
parent(paul,steven).
parent(peter,olivia).
parent(tom,liz).
parent(tony, ann).
parent(michael,paul).
parent(jill,tania).
```

- A *program* is a collection of predicates.
- Clauses within a predicate are used in the order in which they occur.

Prolog Syntax

- Variables begin with a capital letter or underscore:

  X, Socrates, _result

- Atoms do *not* begin with a capital letter:

  socrates, paul

  - Atoms containing special characters, or beginning with a capital letter, must be enclosed in single quotes: 'Socrates'

Prolog

- A *variable* is an identifier beginning with an upper-case letter (e.g., X, Y, Number) or with underscore.
  - *Anonymous variable:* an underscore character (_) stands for 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.
- All types are discovered implicitly (no declarations in LP).

Data types

- An **atom** is a general-purpose name with no inherent meaning.
- **Numbers** can be floats or integers.
- A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms: *tree(node(a),tree(node(b),node(c)))*
  - Special cases of compound terms:
    - *Lists:* ordered collections of terms: [], [1,2,3], [a,1,X|T]
    - *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes. String examples: “abc” (same with the list [97,98,99]), “to be, or not to be”.

Representation of Lists

- Lists are handled as recursive compound terms in Prolog with the first element of the list (called the *head*) and the rest of the list after eliminating the first element (called the *tail*):

  [Head | Tail] OR

  .(Head,Tail)

  - Head is an atom and Tail is a list.
  - We can write [a,b,c] or .(a,.(b,.(c,[]))).

Matching - Unification

- Given two terms, they are identical or the variables in both terms can have same objects after being instantiated

$$date(D,M,2006) = date(D1,feb,Y1)$$

$$D=D1,\ M=feb,Y1=2006$$

- General Rule to decide whether two terms, S and T match are as follows:
  - If S and T are constants, S=T if both are same object
  - If S is a variable and T is anything, T=S
  - If T is variable and S is anything, S=T
  - If S and T are structures, S=T if
    - S and T have same functor
    - All their corresponding arguments components have to match

Declarative and Procedural Way

- The Prolog procedural execution is equivalent with the declarative semantics.

  P:- Q,R.

  - Declarative Way
    - P is true if Q and R are true.
  - Procedural Way
    - To solve problem P, first solve Q and then R (or) To satisfy P, first satisfy Q and then R,
    - Procedural way does not only define logical relation between the head of the clause and the goals in the body, but also the order in which the goal are processed.

Formal Declarative Meaning

- Given a program and a goal G:
- A goal G is true (that is satisfiable, or logically follows from the program) if and only if:
  - There is a clause C in the program such that
  - There is a clause instance I of C such that
    - The head of I is identical to G, and
    - All the goals in the body of I are true.

Evaluation Example

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

sibling(X, Y):- parent_child(Z, X), parent_child(Z, Y).
```

```
?- sibling(sally, erica).
Yes
```
(by chronological backtracking)

Evaluation Example

- **`?- father_child(Father, Child).`**
  enumerates all valid answers on backtracking.

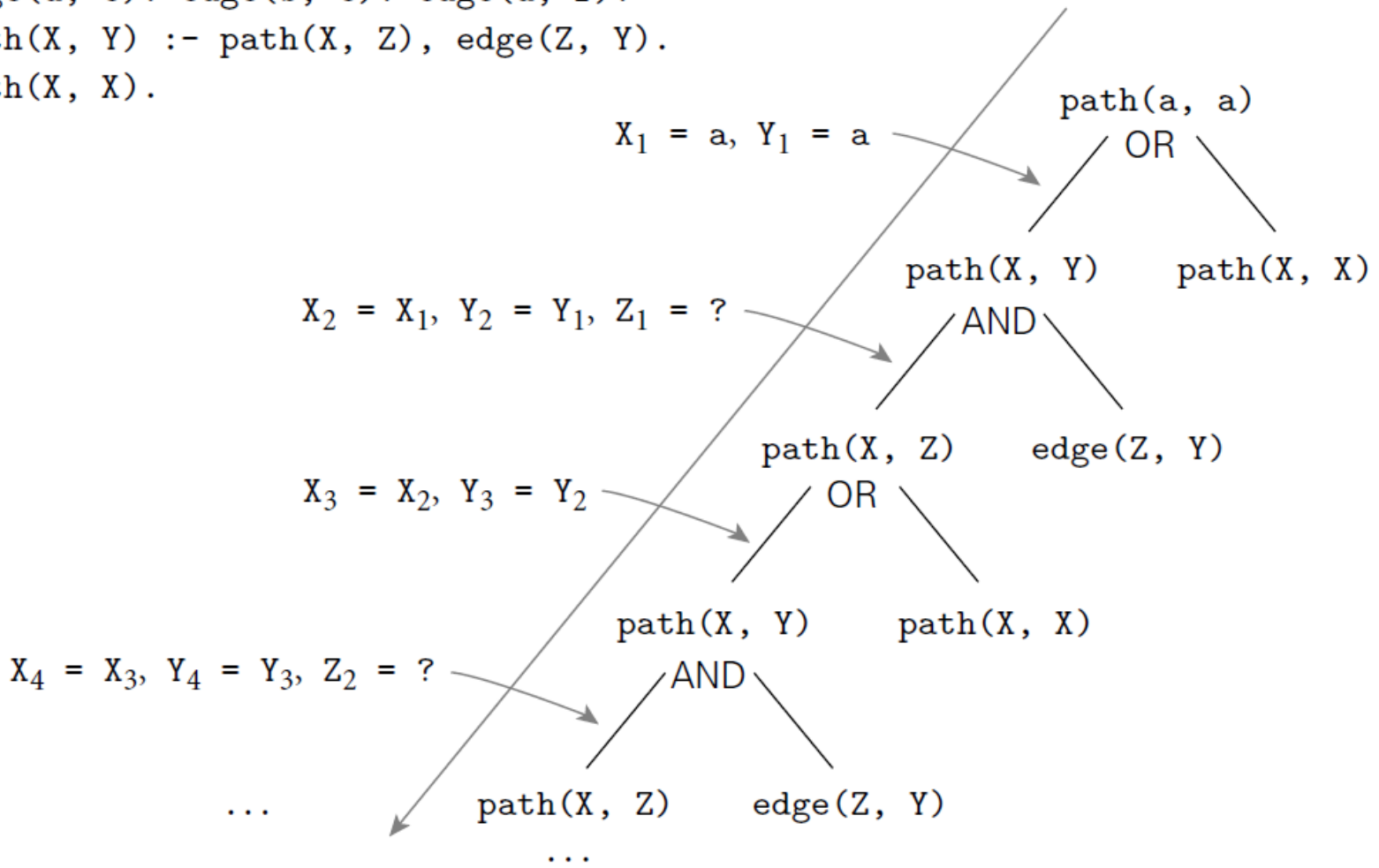
Prolog

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).
```



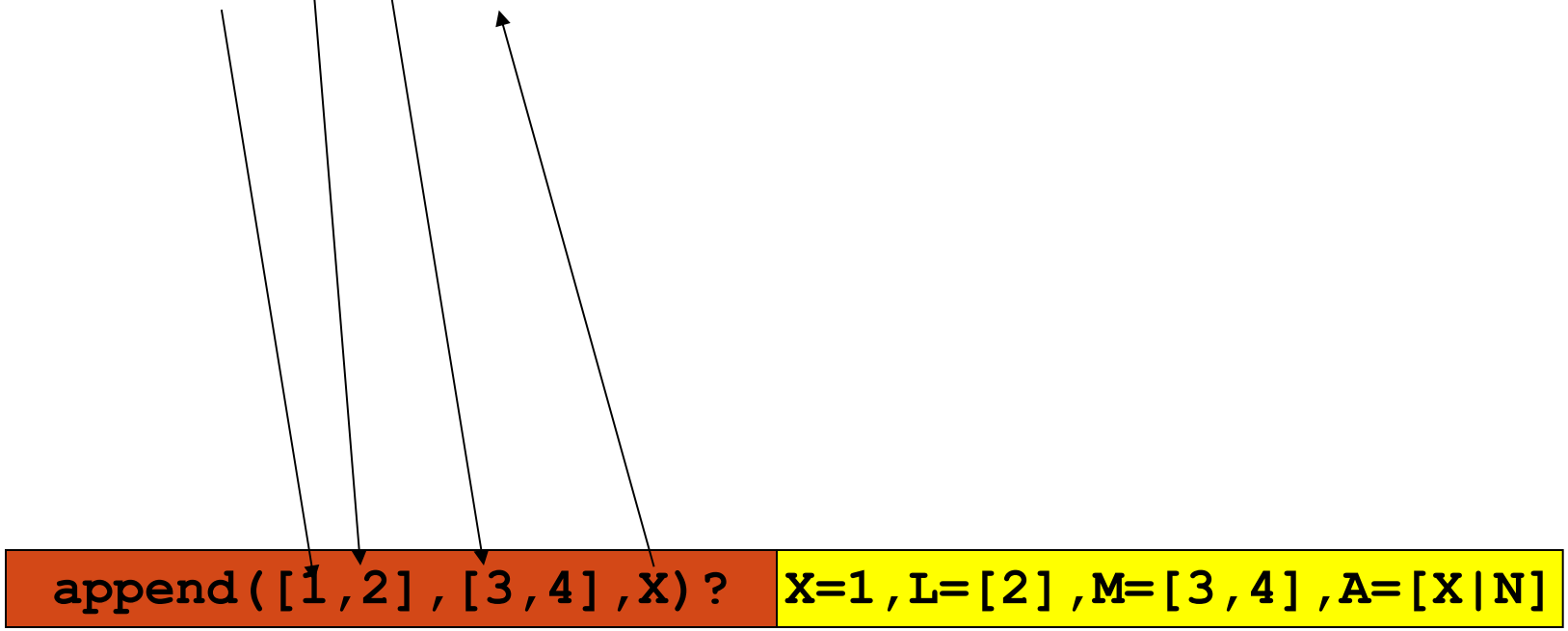
Append example

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

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

Append example

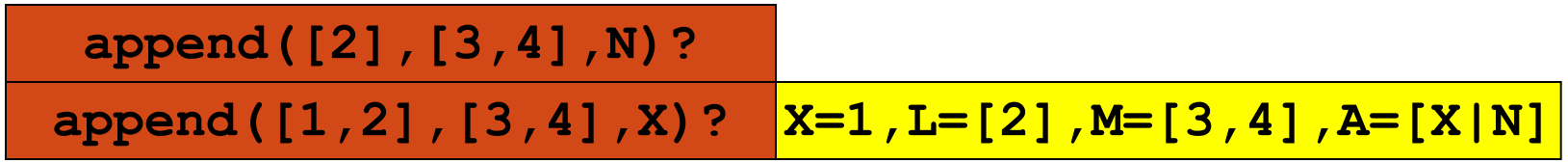
```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```



34

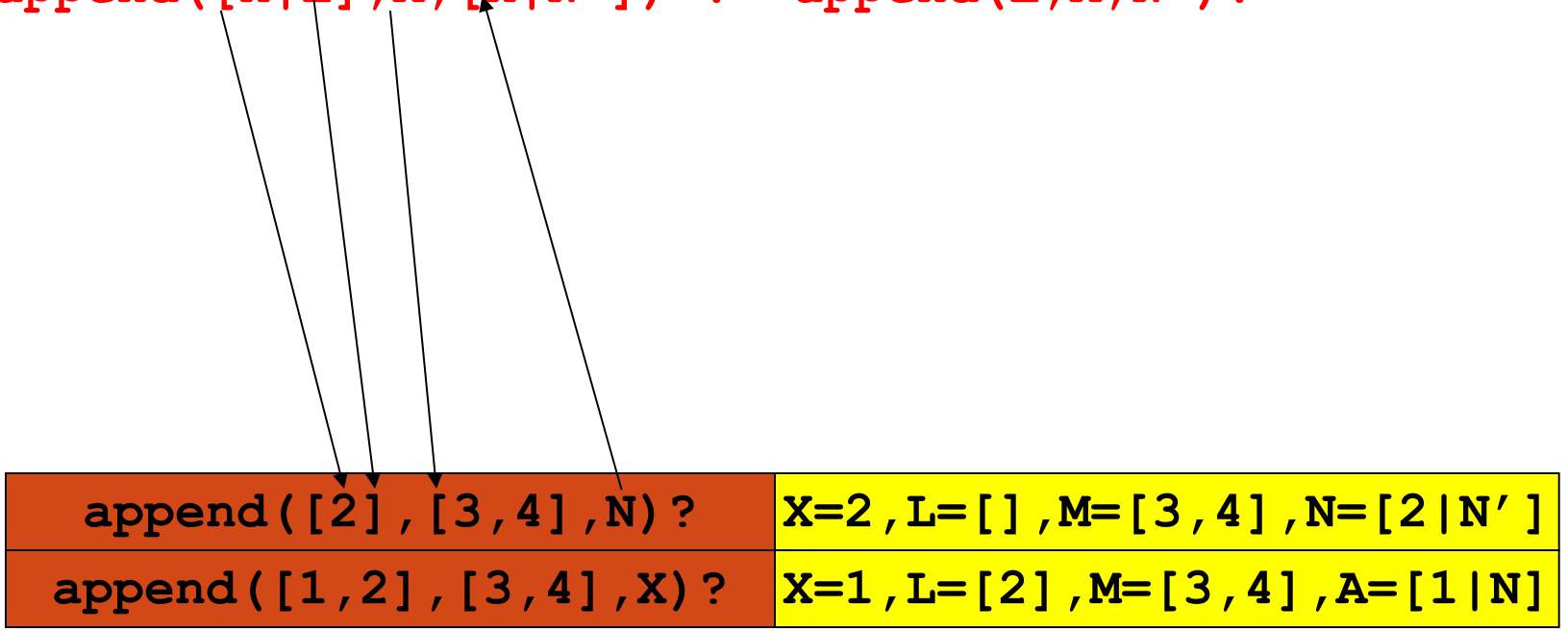
Append example

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



Append example

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



Append example

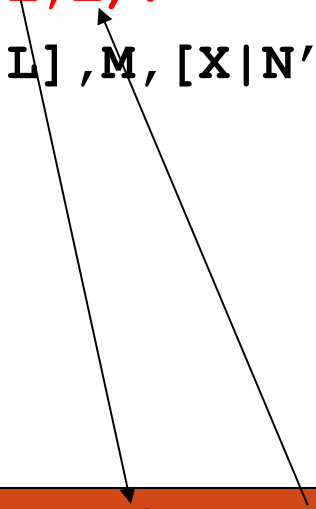
```
append([],L,L).
append([X|L],M,[X|N']) :- append(L,M,N').
```

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

37

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

39

Member Example

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

- *X is a member of a list whose first element is X.*
- *X is a member of a list whose tail is R if X is a member of R.*

```
?- member(2,[1,2,3]).
Yes
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

Select Example

```
select(X,[X|R],R).
select(X,[F|R],[F|S]) :- select(X,R,S).
```

- *When X is selected from [X|R], R results.*
- *When X is selected from the tail of [X|R], [X|S] results, where S is the result of taking X out of R.*

```
?- select(X,[1,2,3],L).
X=1  L=[2,3] ;
X=2  L=[1,3] ;
X=3  L=[1,2] ;
No
```

Reverse Example

```
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).
reverse([],X,X).
```

```
?- reverse([1,2,3],[],X).
X = [3,2,1]
Yes
```

Permutation Example

```
perm([],[]).
perm([X|Y],Z) :- perm(Y,W), select(X,Z,W).
```

```
?- perm([1,2,3],P).
P = [1,2,3] ;
P = [2,1,3] ;
P = [2,3,1] ;
P = [1,3,2] ;
P = [3,1,2] ;
P = [3,2,1]
```

Set Examples

- Sets:

```
union([X|Y],Z,W) :- member(X,Z),  union(Y,Z,W).
union([X|Y],Z,[X|W]) :- \+ member(X,Z), union(Y,Z,W).
union([],Z,Z).

intersection([X|Y],M,[X|Z]) :- member(X,M), intersection(Y,M,Z).
intersection([X|Y],M,Z) :- \+ member(X,M), intersection(Y,M,Z).
intersection([],M,[]).
```

Imperative features

Conditional operator: the if-then-else construct in Prolog:

Example:

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

Difference Lists

- With the default implementation of lists one can only get/put the head of the list in constant time
  - Access to the tail of the list can only be done in a time equal to the size of the list.

Examples:

- Adding an element to the end of a list:

```
addEnd(X,[],[X]).
addEnd(X,[H|T],[H|T2]):- addEnd(X,T,T2).
?- addEnd(6,[1,2,3,4,5],L).
% needs 5 recursion steps to get to the end of the list
%  to add the element
```

- Appending 2 lists requires one to iterate through the first list:

```
append([],L2,L2).
append([H|T],L2,[H|T2]):- append(T,L2,T2).

?- append([1,2,3,4],[5],L3).
```

Difference Lists

- Add element to the end using difference lists:
  - The cost is O(1).
  - The first list A ends with a variable tail B
  - The result is a difference list A-C

```
addEndDL(X, A,B, A,C):- B=[X|C].
?- addEndDL(6,[1,2,3,4,5|B],B,R1,C).
          Response: R1 = [1,2,3,4,5,6|C]
```

- Append with difference lists:
  - The cost is O(1).

```
appendDL(A,AD,B,BD,C,CD):- AD = B, CD = BD, C = A.
?- appendDL([1,2,3,4|AD],AD,[5|BD],BD,C,CD).
          Response: C=[1,2,3,4,5|CD]
```