Notation and Data Structures (cont'd.)

- Can specify head and tail of list using a vertical bar
- Example: `[H|T] = [1, 2, 3]` means `H = 1, T = [2, 3]`
- Example: `[X, Y|Z] = [1, 2, 3]` means `X=1, Y=2, and Z=[3]`
- Built-in predicates include `not`, `=`, and I/O operations `read`, `write`, and `nl` (newline)
- Less than or equal is usually written `=<` to avoid confusion with implication

Execution in Prolog (cont'd.)

- Example 11: clauses entered into database

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
ancestor(X, X).
parent(amy, bob).
```

- Queries:

```
?- ancestor(amy,bob).
yes.

?- ancestor(bob,amy).
no.

?- ancestor(X,bob).
X = amy ->_
```

Execution in Prolog (cont'd.)

- Example 11 (cont'd.): use semicolon at prompt (meaning *or*)

```
?- ancestor(X,bob).
X = amy ->;
X = bob

?-_
```

- Use carriage return to cancel the continued search

Arithmetic (cont'd.)

- Greatest common divisor algorithm
  - In generic Horn clauses:

```
gcd(u, 0, u).
gcd(u, v, w) ← not zero(v), gcd(v, u mod v, w).
```

  - In Prolog:

```
gcd(U, 0, U).
```

Recursion

- `factorial(0,1).`
  `∀ N and ∀ M, factorial(N-1,M) →`
  `factorial(N,N*M).`

- factorial(0,1).
- factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1), F is N * F1.

Tail Recursion

- If the continuation is empty and there are no backtrack points, nothing need be placed on the stack; execution can simply jump to the called procedure, without storing any record of how to come back. This is called LAST-CALL OPTIMIZATION
- A procedure that calls itself with an empty continuation and no backtrack points is described as TAIL RECURSIVE, and last-call optimization is sometimes called TAIL-RECURSION OPTIMIZATION
- Fact(acc,n) {return n==1?acc:Fact(acc*n,n-1)}
- ?-factorial(5,1,F).

Unification

- Unification: process by which variables are instantiated to match during resolution
  - Basic expression whose semantics is determined by unification is equality
- Prolog's unification algorithm:
  - Constant unifies only with itself
  - Uninstantiated variable unifies with anything and becomes instantiated to that thing
  - Structured term (function applied to arguments) unifies with another term only if the same function name and same number of arguments

Unification (cont'd.)

- Examples:

```
?- me = me.
```

```
?- me = you.
```

```
?- me = X.
```

```
?- f(a, X) = f(Y, b).
```

```
?- f(X) = g(X).
```

```
?- f(X) = f(a, b).
```

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

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

Unification (cont'd.)

- Unification causes uninstantiated variables to share memory (to become aliases of each other)
  - Example: two uninstantiated variables are unified

```
?- X = Y.
X = _23
Y = _23
```

- **Pattern-directed invocation**: using a pattern in place of a variable unifies it with a variable used in that place in a goal

```
cons(X, Y, [X|Y]).
```

```
?- cons(0, [1, 2, 3], A).
A = [0, 1, 2,3]
```

Unification (cont'd.)

- Append procedure:

```
append(X, Y, Z) :- X = [], Y = Z.
append(X,Y,Z)   :-
    X = [A|B],
```

- First clause: appending a list to the empty list gives just that list
- Second clause: appending a list whose head is *A* and tail is *B* to a list *Y* gives a list whose head is also *A* and whose tail is *B* with *Y* appended

Unification (cont'd.)

- Append procedure rewritten more concisely:

```
append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).
```

  - Append can also be run backward and find all the ways to append two lists to get a specified list:

```
?- append(X, Y, [1, 2]).
X = []
Y = [1, 2] ->;
```

```
X = [1]
Y = [2] ->;
```

```
X = [1,2].
Y = []
```

Unification (cont'd.)

- Reverse procedure:

```
reverse([], []).
reverse([H|T], L) :- reverse(T, L1),
                     append(L1, [H], L).
```

Unification (cont'd.)

```
gcd(U, 0, U).
gcd(U, V, W) :- not(V = 0) , R is U mod V, gcd(V, R, W).

append([], Y, Y).
append([A|B], Y, [A|W]) :- append(B, Y, W).

reverse([], []).
reverse([H|T], L) :- reverse(T, L1),
                     append(L1, [H], L).
```

Figure 4.1 Prolog clauses for gcd, append, and reverse

Prolog's Search Strategy

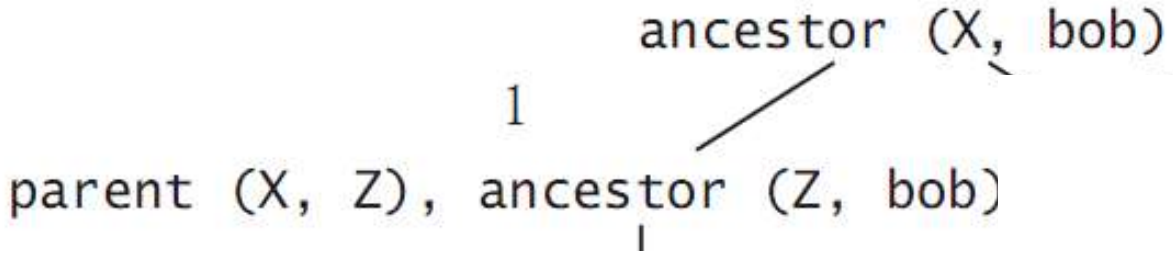
- Prolog applies resolution in a strictly linear fashion
  - Replaces goals from left to right
  - Considers clauses in the database from top down
  - Subgoals are considered immediately
  - This search strategy results in a depth-first search on a tree of possible choices
- Example:

```
(1) ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

(2) ancestor(X, X).

(3) parent(amy, bob).
```

```
(1) ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(amy, bob).
```



Prolog's Search Strategy (cont'd.)

- Leaf nodes in the tree occur either when no match is found for the leftmost clause or when all clauses have been eliminated (success)
- If failure, or the user indicates a continued search with a semicolon, Prolog **backtracks** up the tree to find further paths
- Depth-first strategy is efficient: can be implemented in a stack-based or recursive fashion
  - Can be problematic if the search tree has branches of infinite depth

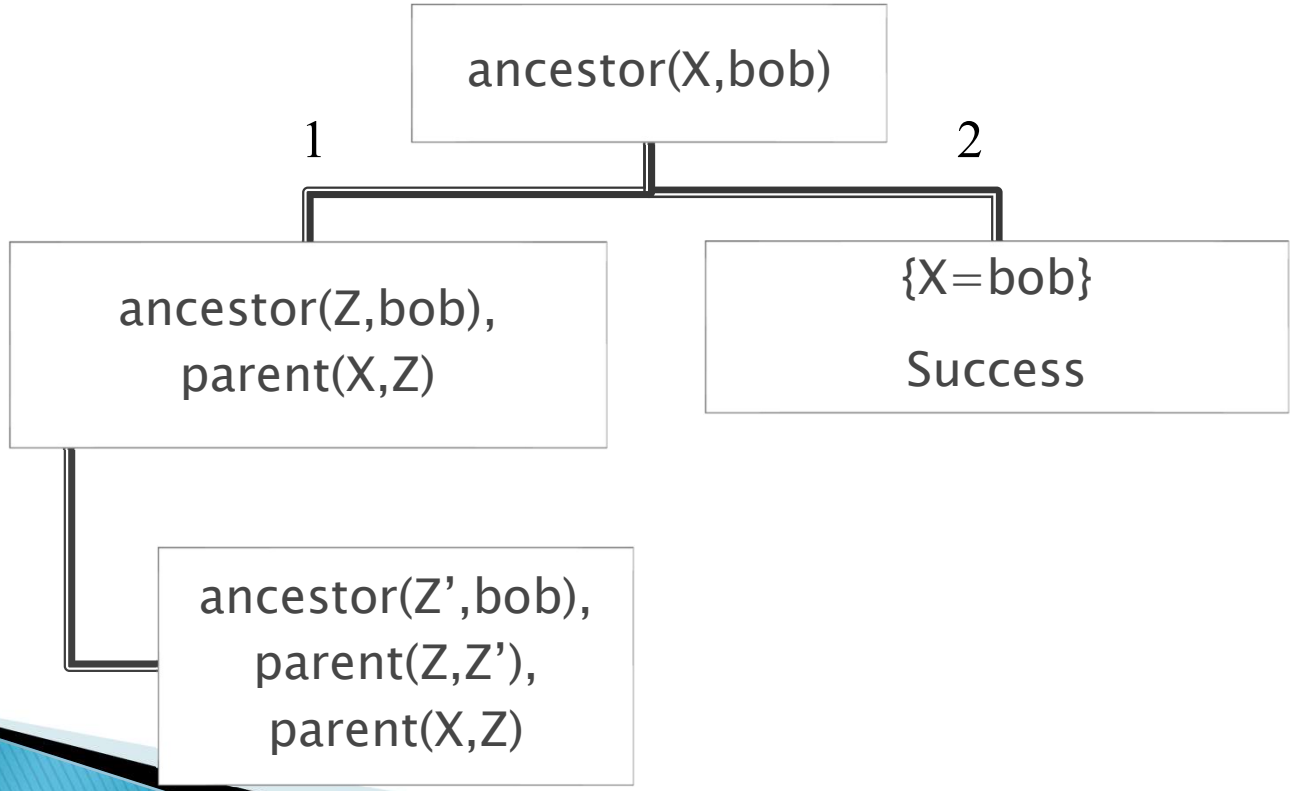
Prolog's Search Strategy (cont'd.)

- Example: same clauses in different order

```
(1) ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).

(2) ancestor(X, X).

(3) parent(amy, bob).
```



Infinite Loop

- Causes Prolog to go into an infinite loop attempting to satisfy *ancestor (Z, Y)*, continually reusing the first clause
- Breadth-first search would always find solutions if they exist
  - Far more expensive than depth-first, so not used

Loops and Control Structures

- Can use the backtracking of Prolog to perform loops and repetitive searches
  - Must force backtracking even when a solution is found by using the built-in predicate *fail*
- Example:

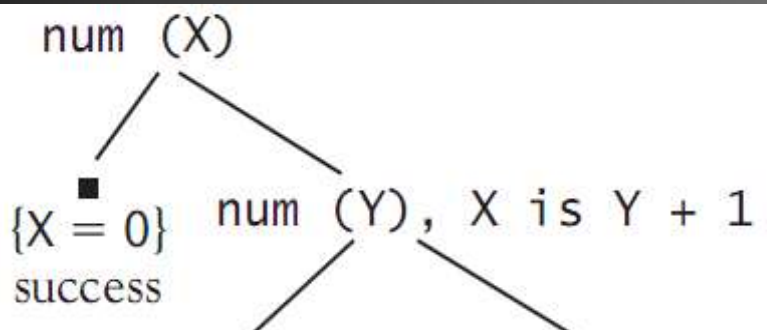
```
printpieces(L) :-append(X, Y, L),
                 write(X),
                 write(Y),
                 nl,
                 fail.
```

Loops and Control Structures (cont'd.)

- Use this technique also to get repetitive computations
- Example: these clauses generate all integers greater than or equal to 0 as solutions to the goal *num(X)*

```
(1) num(0).
(2) num(X) :- num(Y), X is Y + 1.
```

Loops and Control Structures




```
(1) num(0).
(2) num(X) :- num(Y), X is Y + 1.
```

The search tree has an infinite branch to the right

Figure 4.3 An infinite Prolog search tree showing repetitive computations

- Example: trying to generate integers from 1 to 10

```
(1) num(0).
(2) num(X) :- num(Y), X is Y + 1.

writenum(I, J) :- num(X),
                  I =< X,
                  X =< J,
                  write(X),
                  nl,
                  fail.
```

- X=J,!
- Causes an infinite loop after `X = 10`, even though `X =< 10` will never succeed

Max counting

- Predicate max/3 which takes integers as arguments and succeeds if the third argument is the maximum of the first two.

Input	output
?- max(2,3,3)	yes
?- max(3,2,3)	yes
?- max(3,3,3)	yes
?- max(2,3,5)	no
?- max(2,3,X)	Max = 3 yes

1. max(X,Y,Y):- X =< Y.
2. max(X,Y,X):- X>Y.

- There can never be any second solution. So, it should not backtrack.
- The two clauses are mutually exclusive!

1. max(X,Y,Y):- X =< Y,!.
2. max(X,Y,X):- X>Y.

Second clause will be evaluated only if first one does not satisfy. Once got passed the cut, control cannot backtrack!

cut operator (written as !) freezes a choice when it is encountered

Green Cuts:- Cuts like this, which doesn't change the meaning of a program

1. max(X,Y,Y) :- X =< Y,!.
2. max(X,Y,X).

?- max(100,101,X). X=101, yes
?- max(3,2,X). X=3, yes
?- max(2,3,2).

1. max(X,Y,Z) :- X =< Y,!, Y = Z.
2. max(X,Y,X).

Loops and Control Structures

- If a cut is reached on backtracking, search of the subtrees of the parent node stops, and the search continues with the grandparent node
  - Cut prunes the search tree of all other siblings to the right of the node containing the cut
- Example:

```
(1) ancestor(X, Y):- parent(X, Z), !, ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(amy, bob).
```

  - Only `X = amy` will be found since the branch containing `X = bob` will be pruned

Loops and Control Structures (cont'd.)

- Rewriting this example:

```
(1) ancestor(X, Y) :- !, parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X).
(3) parent(amy, bob).
```

Loops and Control Structures (cont'd.)

- Rewriting again:

```
(1) ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
(2) ancestor(X, X) :- !.
(3) parent(amy, bob).
```

  - Both solutions will still be found since the right subtree of `ancestor(X, bob)` is not pruned
  - Search would continue from the grandparent node!

- Cut can be used to reduce the number of branches in the subtree that need to be followed
- Also solves the problem of the infinite loop in the program to print numbers between `I` and `J` shown earlier

Loops and Control Structures (cont'd.)

- One solution to infinite loop shown earlier:

```
num(0).
num(X) :- num(Y), X is Y + 1.
writenum(I, J) :- num(X),
                  I =< X,
                  X =< J,
                  write(X), nl,
                  X = J, !,
                  fail.
```

  - `X = J` will succeed when the upper-bound `J` is reached
  - The cut will cause backtracking to fail, halting the search for new values of `X`

Loops and Control Structures (cont'd.)

- Can also use cut to imitate *if–else* constructs in imperative and functional languages, such as:

  D = if A then B else C

- Prol

```
D :- A, !, B.
D :- C.
```

- Could achieve almost same result without the cut, e executed twice

```
D :- A, B.
D :- not(A), C.
```

Summation of a list

- factorial(0,1).
- factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1), F is N * F1.

- Sum(1,1):-!.
- Sum(N,Res):-

Problems with Logic Programming

- Original goal of logic programming was to make programming a specification activity
  - Allow the programmer to specify only the properties of a solution and let the language implementation provide the actual method for computing the solution
- **Declarative programming**: program describes *what* a solution to a given problem is, not *how* the problem is solved
- Logic programming languages, especially Prolog, have only partially met this goal

Problems with Logic Programming (cont'd.)

- The programmer must be aware of the pitfalls in the nature of the algorithms used by logic programming systems
- The programmer must sometimes take an even lower-level perspective of a program, such as exploiting the underlying backtrack mechanism to implement a cut/fail loop

The Occur-Check Problem in Unification

- **Occur-check problem**: when unifying a variable with a term, Prolog does not check whether the variable itself occurs in the term it is being instantiated to
- Example:

```
is_own_successor :- X = successor(X).
```

- This will be true if there exists an `X` for which `X` is its own successor
- But even in the absence of any other clauses for successor, Prolog answers yes

The Occur-Check Problem in Unification (cont'd.)

- This becomes apparent if we make Prolog try to print such an X:

```
is_own_successor(X) :- X = successor(X).
```

  - Prolog responds with an infinite loop because unification has constructed *X* as a circular structure
  - What should be logically false now becomes a programming error

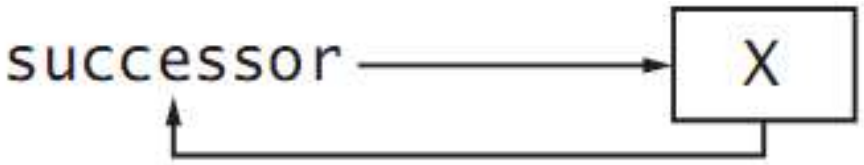


Figure 4-7: Circular structure created by unification

Negation as Failure

- **Closed-world assumption**: something that cannot be proved to be true is assumed to be false
  - Is a basic property of all logic programming systems
- **Negation as failure**: the goal `not(X)` succeeds whenever the goal `X` fails
- Example: program with one clause:
- If we ask `?- not(mother(amy, bob)).parent(amy,bob).`
  - The answer is `yes` since the system has no knowledge of `mother`
  - If we add facts about `mother`, this would no longer be true

Negation as Failure (cont'd.)

- **Nonmonotonic reasoning**: the property that adding information to a system can reduce the number of things that can be proved
  - This is a consequence of the closed-world assumption
- A related problem is that failure causes instantiation of variables to be released by backtracking
  - A variable may no longer have an appropriate value after failure

Negation as Failure (cont'd.)

- Example: assumes the fact *human(bob)*

```
?- human(X).
X = bob

?- not(not(human(X))).
X = _23
```

- The goal `not(not(human(X)))` succeeds because `not(human(X))` fails, but the instantiation of `X` to `bob` is released

Negation as Failure (cont'd.)

- Example:

```
?- X = 0, not(X = 1).
X = 0

?- not (X = 1), X = 0.
no
```

  - The second pair of goals fails because *X* is instantiated to `1` to make `X = 1` succeed, and then `not(X=1)` fails
  - The goal `X = 0` is never reached

Horn Clauses Do Not Express All of Logic

- Not every logical statement can be turned into Horn clauses
  - Statements with quantifiers may be problematic
- Example:

  $p(a)$ and (there exists x, not($p(x)$)).

- Attempting to use Prolog, we might write:

```
p(a).
not(p(b)).
```

  - Causes an error: trying to redefine the *not* operator

Horn Clauses Do Not Express All of Logic (cont'd.)

- A better approximation would be simply *p(a)*
  - Closed-world assumption will force *not(p(X))* to be true for all *X* not equal to *a*
  - But this is really the logical equivalent of:

$$p(a) \text{ and } (\text{for all } x, \text{not}(x = a) \rightarrow \text{not}(p(a))).$$

  - This is not the same as the original statement

Control Information in Logic Programming

- Because of its depth-first search strategy and linear processing of goals and statements, Prolog programs also contain implicit information on control that can cause programs to fail
  - Changing the order of the right-hand side of a clause may cause an infinite loop
  - Changing the order of clauses may find all solutions but still go into an infinite loop searching for further (nonexistent) solutions

Control Information in Logic Programming

- One would want a logic programming system to accept a mathematical definition and find an efficient algorithm to compute it
- Instead, we must specify actual steps in the algorithm to get a reasonable efficient sort
- In logic programming system, we not only provide specifications in our programs, but we must also provide algorithmic control information

Insertion Sort

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

insortx(X, [A|L], [A|M]) :-
        order(A, X), !, insortx(X, L, M).
insortx(X, L, [X|L]).
```

1. insert(X,[],[X]).

2. insert(X,[H|Tail],[X,H|Tail]):- X =< H.

3. insert(X,[H|Tail],[H|NewTail]):- X > H,
   insert(X,Tail,NewTail).

1. isort([],[]).

2. isort([X|Tail],SList):- isort(Tail,STail),
   insert(X,STail,SList).

Quick sort

For the recursive rule, we first remove the Head from the unsorted list and split the Tail into those elements preceding Head wrt. the ordering Rel (list Left) and the remaining elements (list Right). Then Left and Right are being sorted, and finally everything is put together to return the full sorted list

```
quicksort(Rel, [Head|Tail], SortedList) :-
  split(Rel, Head, Tail, Left, Right),
  quicksort(Rel, Left, SortedLeft),
  quicksort(Rel, Right, SortedRight),
  append(SortedLeft, [Head|SortedRight], SortedList).
```

Quick Sort

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

split(Rel, Middle, [Head|Tail], [Head|Left], Right) :-
  check(Rel, Head, Middle), !,
  split(Rel, Middle, Tail, Left, Right).

split(Rel, Middle, [Head|Tail], Left, [Head|Right]) :-
  split(Rel, Middle, Tail, Left, Right).
```

Quick Sort

```
qsort([], []).
qsort([H|T], S) :- split(H, T, L, R),
                   qsort(L, L1),
                   qsort(R, R1),
                   append(L1, [H|R1], S).
split(P, [A|X], [A|Y], Z) :- A < P,
                             split(P, X, Y, Z).
split(P, [A|X], Y, [A|Z]) :- A >= P,
                             split(P, X, Y, Z).
split(P, [], [], []).
```