More on Prolog syntax

- Already seen program statement types:
  - rules: p(X) :- q(X,Y), r(Y,z).
  - facts: likes(brian, madonna).
  - queries: ?- likes(madonna, brian).

- Argument forms:
  - logical variables: identifiers starting with upper-case letters

    eg. Guy, X, List34...
  - constants: identifiers starting with lower-case, integers, numbers, operators

    eg. brian, c, f45, long_name, 34, 5.5 ...

    also: single quotes 'XYZ' (not commonly done)
  - structures: a constant with arguments
    - » let you create more "complex" data, possibly with variable components
    - » look like goals or predicate names

    eg. author(X, Book)   tree(Left, Right)

    a(b(C, D), e(f, G))   tree(tree(Left, Right), Right2)

    XXXXX   illegal: X(a, b)XXXXXX

  - structures can be nested arbitrarily deep

Prolog syntax: operators, lists

- Prolog lets you define structures in which the structure name looks more like an algebraic operator
  - eg. infix: D + 4 --> shorthand for +(D, 4)
  - prefix: \+ G --> \+ (D)
  - postfix: G++ --> ++ (D)

- list: special builtin structure

  [ ] --> empty list; shorthand for '.' ( )

  [H | T] --> first element is H, tail is list T

  - shorthand for '.' (H, T)

  [a, b, c, d] = [H | T] --> H = a, T = [b, c, d]

  - can have nested lists too: [[a, b], c, [d, e], [ ], f]

  *** exercise: write this nested list in it's "full" notation ***

Basic Prolog execution

- We start with a program query(“Q”), that has the form

?- G1, G2, ... , Gk. where k >= 1,
Gi = pred(Arg1, ..., Arg_n), n > 0

- Prolog interpreter tries to solve Q
- Basically, interpreter tries to reduce Q to an empty list of goals
  - when empty --> a solution has been found, and it is returned to user
  - otherwise, either the program fails (no solution) or does not terminate (the query doesn’t shrink)
- standard Prolog execution strategy:
  1. <u>computation rule</u> (or goal selection rule): goals in query are solved from left--to--right
  2. <u>search rule</u> (or clause selection rule): program clauses are matched or <u>unified</u> with query goals in the order they’re found in the program
  3. <u>backtracking</u>: when there is no clause that matches, go back to the last place you chose a clause, and try the next one.

Unification

- **unification: Prolog's matching technique**
  - **type of symbolic pattern matching**
  - **given two terms, finds a set of variable substitutions which, when applied to each term, results in the same term**
  - **finds most general substitions (least specific same term)**

- **Unification algorithm (see page 40 Bratko):**
  - **1. an uninstantiated (ie. unbound) variable unifies with anything**
  - **2. a constant or integer will unify only with itself**
  - **3. a structure will unify with another structure only if:**

    a) it has the same structure name and number of arguments

    b) all the corresponding arguments unify (ie. recursively call unify on them!)

- **= is the builtin call to unification algorithm**
- **Remember: logic variables are placeholders for constants, structures**
  - **they therefore unify with anything**

Example unifications

1. apples = apples --> yes (case 2)
2. apples = 4 --> no (case 2)
3. X = cat --> yes (case 1)
4. X = dog(Y) --> yes (case 1)
5. Apples = Oranges --> yes (case 1)
6. mouse(a, X) = mouse(D, 2) --> yes { D <- a, X <- 2 }
7. cat(a, B, Y) = cat(A, X, c(d)) --> yes: { A <- a, B <- X, Y <- c(d) }
8. dog(a, b) = dog (A, B, c) --> no: different # arguments
9. m(A, b, A) = m(1, b, 2) --> no: A cannot unify with both 1 and 2
10. [1, 2, 3, 4] = [H | T] --> yes: { H <- 1, T <- [2, 3, 4] }
11. [1, 2, 3, 4, [5, 6]] = [A, B | C] --> yes: { A <- 1, B <- 2, C <- [3,4,[5,6]]}
12. [1, 2] = [A, B, C] --> no: different number of arguments
13. likes(brian, cindy_crawford) = likes(cindy_crawford, brian) --> no :-(

Applying unifying substitutions

- the unification algorithm’s output is a set of variable bindings
  - set might be empty!
- from previous slide, if we apply a substitution to each term, we get the identical term

eg. 6. mouse(a, X) = mouse(D, 2) --> yes { D <- a, X <- 2 }

--> mouse(a, 2)

eg. 7. cat(a, B, Y) = cat(A, X, c(d)) --> yes: { A <- a, B <- X, Y <- c(d) }

---> cat(a, B, c(d))

Unifying during execution

```
?-   p(a, X, b(Y), 1).

p(b, A, B, C).                    %1
p(a, b, Z, W).                    %2
p(A, B, A, C).                    %3
p(a, X, b(25), C) :- q(X, 44), r(C).    %4
p(_, d, Y, _).                    %5

p(1, 2, 3).                       %6
```

- treat the goal and predicate as structures to unify, and apply unification to them
- note how data passes in two directions during unification
  - eg. case 2:{ X <- b, Z <- b(Y) }
  - this permits very powerful predicates: arguments can supply data <u>to</u> predicate, OR return computed data <u>from</u> predicate

More example unifications

1. foo(1, foo(2,3)) = foo(1,Z) --> yes: Z = foo(2,3)
2. ‘.’(1, ‘.’(2, 3)) = ‘.’(1, Z) --> yes: Z = ‘.’(2, 3)
3. [1 | [2, 3]] = [1 | Z] --> yes: Z = [2, 3]
4. [1, [2, 3]] = [1, 2, 3] --> no: [1, [2,3]] is ‘.’(1, ‘.’(‘.’(2, ‘.’(3, []))))
   [1, 2, 3] is ‘.’(1, ‘.’(2, ‘.’(3, [])))
5. [a,b,c] = [a| [b,c]] -> yes: and = [a, b | [c]] = [a, b, c | []]
6. [a, b, c] = [A] --> no
7. [a, [b, c]] = [X, Y] --> yes: X = a, Y = [b, c]
8. [a | [b, c] = [X, Y] --> no: LHS same as [a, b, c]
9. [a | [b | [c]]] = [X | Y] --> yes: X = a, Y = [b, c]
10. [a, b, c] = [X, Y, Z] --> yes: X = a, Y = b, Z = c
11. [a, b, c] = [X, Y, X] --> no: if X = a, then X \= c
12. [[a], b] = [X|Y] --> yes: X = [a], Y = [b]
13. [[a], [b]] = [X|Y] --> yes: X = [a], Y = [[b]]
14. [_] = [X|Y] --> yes: X = _, Y = []

Back to Prolog execution

LOOP until Q is empty OR cannot find a solution:

- take first goal G_1 in Q; (eg. G_1 = p(A1, ..., An))
- unify G_1 with a clause in the predicate p;
  - » try each clause in the order found in predicate
  - » a) if no clauses unify --> BACKTRACK
  - » b) otherwise, for the first clause that unifies:
    - let the clause be p(B1,...,Bn) :- B1, ... , Bm. (m >= 0)
      - (refresh variable names in it)
    - let the binding substition set be Ω
    - replace G1 in query with B1,...,Bm
    - apply Ω to the whole query

- note: when a rule is used, the query can grow in size
  while facts cause the query to shrink in size

- BACKTRACK: this causes interpreter to go back to the last place where a choice of clauses was made, and to try next one

Execution trees

- It is difficult to conceptualize the execution of Prolog when backtracking and recursion is occurring.

- But Prolog program execution can be easily “visualized” by using an execution tree (aka search tree)

(a) root of tree = program query

(b) non-leaf nodes of trees =intermediate computed queries

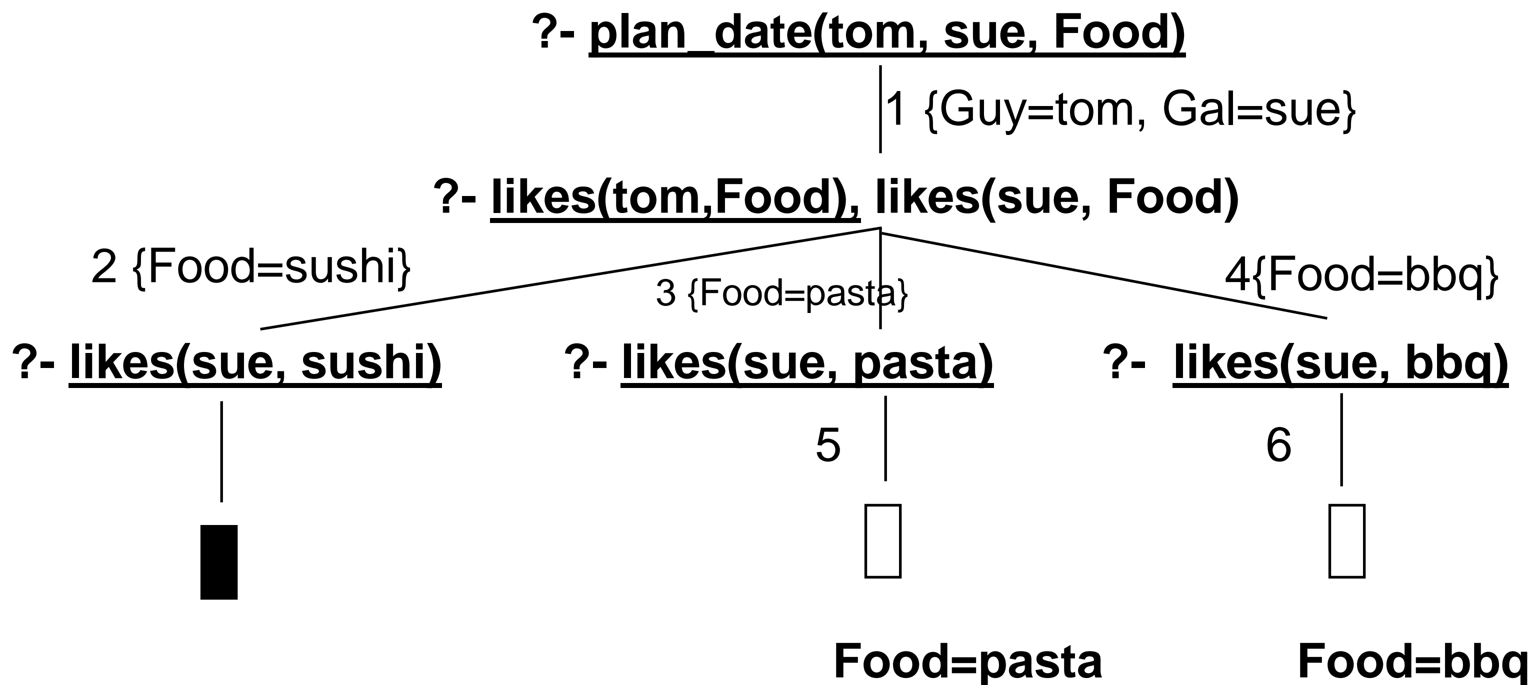
(c) branch = the unification of 1st goal of parent node with a clause
  - can label branch with (i) clause selected, and (ii) variable substitutions used in unification
  - order of branches from left to right reflects order of clauses in program

(d) leaf nodes: either success □ or failure ■
  - success leafs: can give final substitutions of goal variables
  - failure leafs: represent places when interpreter does backtracking

Execution trees

```
1:    plan_date(Guy, Gal, Food) :- likes(Guy, Food), likes(Gal, Food).

2:    likes(tom, sushi).
3:    likes(tom, pasta).
4:    likes(tom, bbq).
5:    likes(sue, pasta).
6:    likes(sue, bbq).
```



{Food <- pasta }

{ Food <- bbq }

Backtracking

•Variables are unique within each clause.

• always rename variables in clauses so that they don't clash with those in the current goal

eg.

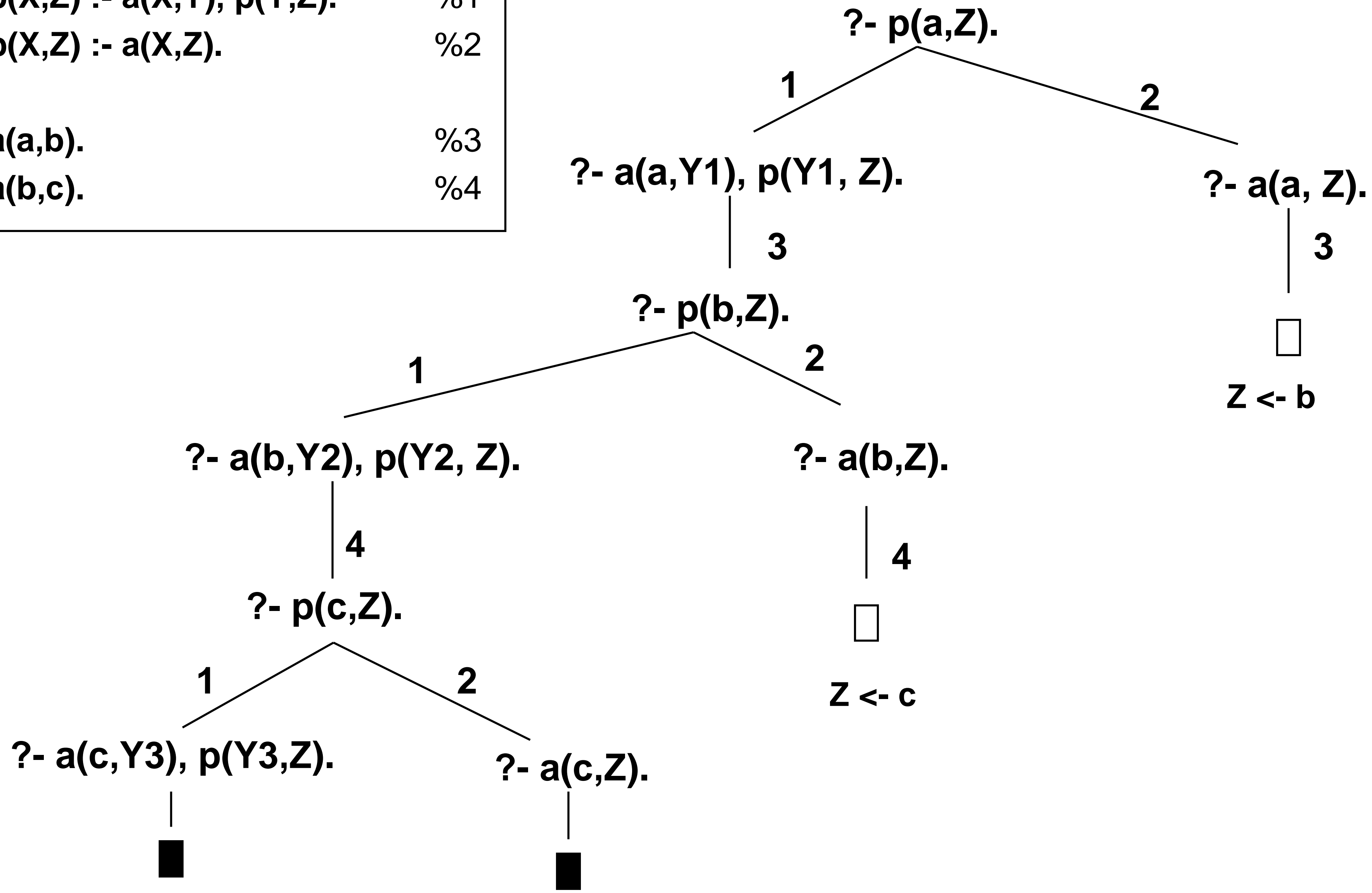
```
  grandparent(A,B)  :-  parent(A,X), parent(X,B).

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

```
?-  grandparent(X, tom).
  |
  |    ( grandparent(A,B) :- parent(A, X' ) , parent(X' , B).     )
  |         = { X <- A, B <- tom }
  v

?- parent(A, X'), parent(X', tom).
```

Another execution tree



Some possible program behaviors

1. no solutions: output --> "no"

eg. ?- pet(cat).

```
pet(dog).
```

2. a finite number of solutions

eg. ?- pet(X).

```
pet(cat).
pet(dog).
```

3. an infinite number of solutions

eg. ?- pet(Y).

```
pet(dog).
pet(X) :- pet(X).
```

4. non-termination, and eventually memory overflow

eg. ?- pet(X).

```
pet(Y) :- pet(Y).
```

5. computation error: bad use of builtin predicates

6. bad solution: eg. pet(television). (erroneous logic)

Arithmetic

- Note that arithmetic expressions in Prolog are simply structures

1+X*Y-4   --> - (+ (1, * (X, Y)), 4)

- Unification (=) uses symbolic structures to unify terms
- Hence, unification does not see the arithmetic values of expressions

  2 + 3 * 4 = 14   --> fails!

- <u>is</u> : builtin arithmetic equality operator
  - form: X is Expression
  - whereX is either a variable (instantiated or not) or a constant
  - Expression is an arithmetic expression
  - all variables in Expression MUST be unified, else a run-time error

- 'is' (i) evaluates Expression; (ii) unifies value of expression with X
- (hence X is a variable or constant)

Arithmetic

?- X is 2 + 5.

X = 7

?- Y = 4, X is 3 * Y.

X = 12.

?- X is 3 * Y.

{ INSTANTIATION ERROR: in expression }

?- Y is (6 + 6) / 3.

Y = 4.0

?- 4.0 is (6 + 6) / 3.

yes

?- 4 is (6 + 6) / 3.

no   (because Prolog converts expressions with '/' to float)

?- 6 + 6 is 3 * 4.

no   (because 6+6 is a structure)

?- cat is 2 + 2.

no   (cat \ = 4 !)