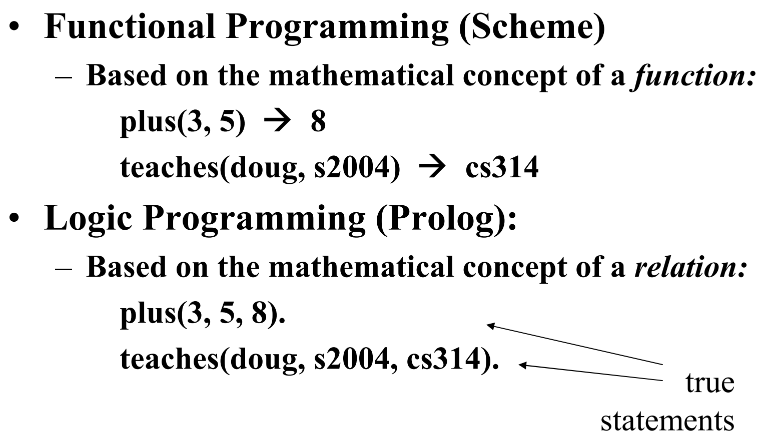
Announcements

- C++ project on web today
- Short homework on parameter passing also will be there (very soon) due next week
- No Thursday office hours this week
- Final exam date was wrong on web page (is actually May 10th, not 12th)

1

Scheme vs. Prolog

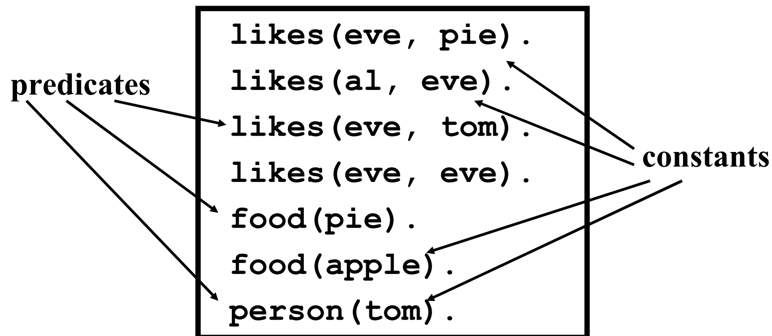
- Functional Programming (Scheme)
  - Based on the mathematical concept of a *function:*
    - plus(3, 5) → 8
    - teaches(doug, s2004) → cs314
- Logic Programming (Prolog):
  - Based on the mathematical concept of a *relation:*
    - plus(3, 5, 8).
    - teaches(doug, s2004, cs314).



2

Prolog

- **Database of Facts:**



Prolog

- **Queries:**

```
?-likes(al,eve).
yes
?-likes(al, pie).
no
?-likes(eve,al).
no
?-likes(person,food).
no
```

query

answer

```
likes(eve, pie).
likes(al, eve).
likes(eve, tom).
likes(eve, eve).
food(pie).
food(apple).
person(tom).
```

Prolog

- **Queries:**

variable

```
?-likes(al,Who).
Who=eve
?-likes(eve,W).
W=pie ;
W=tom ;
W=eve ;
no
```

answer with variable binding

force search for more answers

```
likes(eve, pie).
likes(al, eve).
likes(eve, tom).
likes(eve, eve).
food(pie).
food(apple).
person(tom).
```

5

Prolog

- **More Complex Queries:**

```
?-likes(A,B).
A=eve,B=pie ;
A=al,B=eve ; …
?-likes(D,D).
D=eve ;
no
?-likes(eve,W), person(W).
W=tom
?-likes(al,V), likes(eve,V).
V=eve ;
no
```

and

```
likes(eve, pie).
likes(al, eve).
likes(eve, tom).
likes(eve, eve).
food(pie).
food(apple).
person(tom).
```

6

Prolog

- **More Complex Queries:**

```
?-likes(eve,W),likes(W,V).
W=eve,V=pie ;
W=eve,V=tom ;
W=eve,V=eve
?-likes(eve,W),person(W).
W=tom
?-likes(eve,V),(person(V);food(V)).
V=pie  ; V=tom ; no
?-likes(eve,W), \+ likes(al,W).
W=pie ; W=tom ; no
```

same binding

or

not

```
likes(eve, pie).
likes(al, eve).
likes(eve, tom).
likes(eve, eve).
food(pie).
food(apple).
person(tom).
```

7

Prolog

- **To abbreviate a query:**

```
?-q1.
yes
?-q2(H).
H=tom ;
no
?-q2(pie).
no
```

```
likes(eve, pie).
likes(al, eve).
likes(eve, tom).
likes(eve, eve).
food(pie).
food(apple).
person(tom).
```

Database

```
q1  :- likes(eve,V), person(V).
q2(V)  :- likes(eve,V), person(V).
```

Rulebase

- **Add a Rule:**

8

Prolog

- **Rules are used to define new predicates:**
  - **`q1` is a predicate with no arguments, i.e., a proposition, which might be called `eveLikesSomeone`.**
  - **`q2` is a predicate with one argument, which might be called `personLikedByEve`.**

```
eveLikesSomeone :- likes(eve,V), person(V).
personLikedByEve(V) :- likes(eve,V), person(V).
```

  - **Similarly, we could define the predicate `likeable` as:**

```
likeable(V) :- likes(W,V), person(V).
```

9

Logic Programming

- **Declarative Semantics (based on predicate logic):**
  - **constants: `eve, tom, pie, 3, ...`**
  - **variables: `Who, W, V, ...`**
  - **atomic formulae: `likes(e,t), likes(W,t), ...`**
  - **Horn clauses: `q(X,Y) :- r(X,W,Y,b), s(W,3).`**
    $(\forall$ `X`$)$ $(\forall$ `Y`$)$ [`q(X,Y)` $\Leftarrow$ $(\exists$ `W`$)$[`r(X,W,Y,b)` & `s(W,3)`]]
  - **Logic Program: a set of Horn clauses, either facts (with antecedent `true`) or rules (as shown above).**
  - **Query or Goal: the antecedent of a rule.**
  - **Are there bindings for the free variables in the Goal that would make it logically entailed by the Program?**

10

Logic Programming

- **Procedural Semantics:**
  - **Interpret a predicate as a boolean-valued function.**
  - **Interpret the Horn clause**

    `p(X,Y):- q(X,W), r(Y,W,2).`

    as part of the definition of the function `p`, so that a call to `p(a,1)` results in a call to `q(a,W)` and `r(1,W,2)`.
  - **Each call to the function `p` tries to find**
    - **a rule or fact defining `p`, and**
    - **a set of values for its free variables, such that the result returned by the function `p` is `true`.**
  - **This requires *search* and *unification* of the free variables.**

Search Trees

```
?-liked(eve).
```

```
liked(eve)
   |  V'= eve
likes(W',eve)
```

```
likes(eve,pie).
likes(al,eve).
likes(eve,tom).
likes(eve,eve).
food(pie).
food(apple).
person(tom).
```

```
liked(V) :- likes(W,V).
```

Search Trees

```
?-liked(eve).
```

```
liked(eve)
  V'= eve
likes(W',eve)
  W'= al
success
```

```
likes(eve,pie).
likes(al,eve).
likes(eve,tom).
likes(eve,eve).
food(pie).
food(apple).
person(tom).
```

```
liked(V) :- likes(W,V).
```

13

Search Trees

```
?-liked(X).
```

```
liked(X)
  X=V'
likes(W',V')
  W'=eve,V'=pie
success
```

```
likes(eve,pie).
likes(al,eve).
likes(eve,tom).
likes(eve,eve).
food(pie).
food(apple).
person(tom).
```

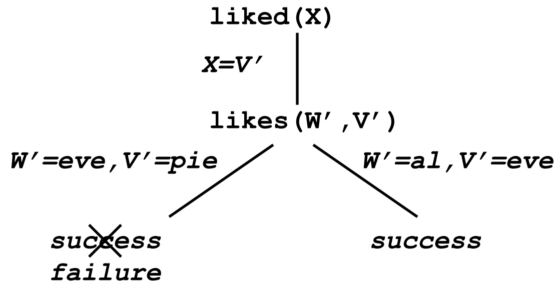
```
X = pie
```

```
liked(V) :- likes(W,V).
```

14

Search Trees

`?-liked(X).`




```
likes(eve,pie).
likes(al,eve).
likes(eve,tom).
likes(eve,eve).
food(pie).
food(apple).
person(tom).
```

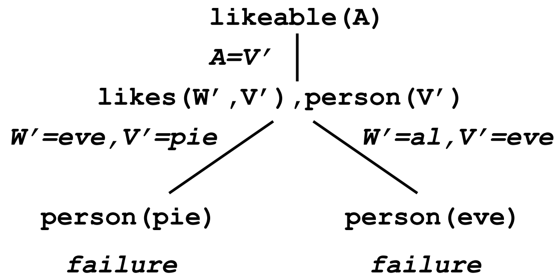
```
X = pie ;
X = eve
```

```
liked(V) :- likes(W,V).
```

15

Search Trees

`?-likeable(A).`



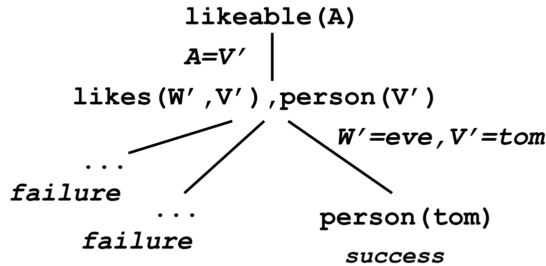

```
likes(eve,pie).
likes(al,eve).
likes(eve,tom).
likes(eve,eve).
food(pie).
food(apple).
person(tom).
```

```
likeable(V) :- likes(W,V),person(V).
```

16

Search Trees

`?-likeable(A).`




```
likes(eve,pie).
likes(al,eve).
likes(eve,tom).
likes(eve,eve).
food(pie).
food(apple).
person(tom).
```

`A = tom`

```
likeable(V) :- likes(W,V),person(V).
```

17

Search Order

- In the “pure” logic programming paradigm, search order is nondeterministic.
- But in Prolog, search order is deterministic:
  - Facts and rules are searched in the order in which they are written.
  - Conjunctive goals are processed from left to right.
  - The search tree is traversed in depth-first order, with backtracking to the previous choice point.
- The programmer must keep this execution model in mind! (Example: `likeable.pl`)

18

Recursive Rules

- **Family Relations:**

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

- **This is incorrect:**

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

- **This is correct:**

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

- **Why? (Example: `family.pl`)**

Equality and Inequality

- **To define siblings:**

```
sibling(X,Y) :- mother(M,X), mother(M,Y),
                X \== Y,
                father(F,X), father(F,Y).
halfsibling(X,Y) :- parent(Z,X), parent(Z,Y),
                    X \== Y,
                    \+ sibling(X,Y).
```

- **Terms are identical (or not):**

  `X == Y` **or** `X \== Y`

- **Terms are unifiable (or not):**

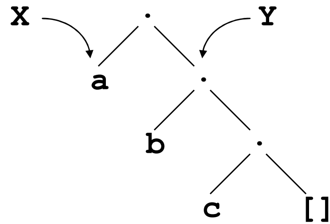
  `X = Y` **or** `\+ X = Y`

Data Structures: Lists

- **List syntax:**

```
| ?- [a,b,c] = [X|Y].
X = a, Y = [b,c] ? ; no

| ?- [[the,cat],sat] = [X|Y].
X = [the,cat], Y = [sat] ? ; no
| ?- [the,[cat,sat]] = [X|Y].
X = the, Y = [[cat,sat]] ? ; no
| ?- [cat] = [X|Y].
X = cat, Y = [] ? ; no
| ?- [] = [X|Y].
no
```



Data Structures : Lists

- **Unifying Lists:**

```
| ?- [X,Y,Z] = [john,likes,fish].
X = john, Y = likes, Z = fish ? ; no

| ?- [[the,Y]|Z] = [[X,hare],[is,here]].
X = the, Y = hare, Z = [[is,here]] ? ; no

| ?- [X,2|W] = [1,2].
W = [], X = 1 ? ; no

| ?- [X,2|W] = [1,2,5,4].
W = [5,4], X = 1 ? ; no
```

Predicates on Lists

- **Appending two lists:**

```
app([],L,L).
app([X|L1],L2,[X|L3]) :- app(L1,L2,L3).
```

- **What result?**

```
| ?- app([1,2],[3,4],Z).
| ?- app([1,2],Y,[1,2,3,4]).
| ?- app(X,[3,4],[1,2,3,4]).
| ?- app(X,[4,3],[1,2,3,4]).

| ?- app(X,Y,[1,2,3,4]).
| ?- app(X,Y,Z).
```