Logical Languages

part 3

2020

Instructor: Odelia Schwartz

Prolog

To access the lab computers, ssh into johnston and then ssh into one of the host computers in the lab. To see what hosts are available type in the johnston command line cat ~irina/hostnames

There are a couple of files available for download from the class website, such as simple.pl.

Prolog

:-   implies symbol
,   and symbol

- Right side implies left side
  Right side can have and

- Headless or headed

- Facts
  Rules
  Goals/Queries

- Variables: start with capital letter

Prolog

Prolog demos

simple.pl file includes:

```
% Simple example for testing
% swipl from command line
% Inside compiler:
% ['simple.pl'].
% person(bob).
% returns true
% father(bob,X).
% returns X = sam.
% control d to exit

person(bob).
father(bob,sam).
```

Prolog

Prolog demos

simple.pl let’s try it in compiler:

- swipl from command line

- Inside compiler:
  ['simple.pl’].

- control d to exit

Notice we always have a period after statement

Prolog

Prolog demos

simple.pl let’s try it in compiler:

- swipl from command line

- Inside compiler:
  ['simple.pl’].

Things to try:
  person(bob).
  father(bob,X).

Prolog

Prolog demos

simple.pl let's try it in compiler:

- swipl from command line

- Inside compiler:
  ['simple.pl'].

Things to try:

person(bob). Returns true
father(bob,X). Returns X=sam

Prolog

Prolog demos

simplemore.pl let’s add more facts to file:

```
person(bob).
father(bob,sam).
father(sam,liz).
```

father(bob,X).
Returns?

Prolog

Prolog demos

simplemore.pl let’s add more facts to file:

```
person(bob).
father(bob,sam).
father(sam,liz).
```

father(bob,X).
Returns?

initially returns X = sam
Type ; and will return next item here:
X = liz

Prolog demos

Let's try simple2.pl

```
%http://faculty.otterbein.edu/psanderson/csc326/notes/PrologNotes.html

mother(iva, pete).
mother(iva, ed).
mother(iva, becky).
mother(kay, nancy).
mother(kay, bob).
mother(kay, diane).
mother(becky, katie).
husband(dwight, iva).
husband(robert, kay).
husband(pete, nancy).

wife(X,Y) :- husband(Y,X).
father(X,Y) :- husband(X,Z), mother(Z,Y).
```

Things to query:
mother(kay, nancy).
mother(kay, kay).
mother(kay, Who). press ;

Prolog

Inferencing process of Prolog. Example:

man(bob) query

Database includes rules:
father(bob).
man(X) :- father(X).

How does Prolog do it? Two possibilities:

1. Forward chaining: search for and find first proposition father(bob); goal is inferred by matching first proposition with right side of second rule father(X) through instantiation of X to bob, and then matching left side of second proposition to goal man(bob)

Prolog

Inferencing process of Prolog. Example:

man(bob) query

Database includes rules:
father(bob).
man(X) :- father(X).

How does Prolog do it? Two possibilities:

2. Backward chaining: first match goal with left side of second proposition man(X) through the instantiation of X to bob; as last step, match right side of second proposition (now father(bob)) with first proposition

Prolog

How does Prolog do it? Two possibilities:

1. Forward chaining: search for and find first proposition father(bob); goal is inferred by matching first proposition with right side of second rule father(X) through instantiation of X to bob, and then matching left side of second proposition to goal man(bob)

2. Backward chaining: first match goal with left side of second proposition man(X) through the instantiation of X to bob; as last step, match right side of second proposition (now father(bob)) with first proposition

Prolog uses Backward chaining. First match goal.

Prolog

Backtracking

- Multiple subgoals
- If fail to show proof of one subgoal, reconsider previous subgoal to find alternative solution (backtracking)
- Begin search where previous search left off
- Can take lots of time and space, because may find all possible proofs for every subgoal

Prolog

Backtracking

- Multiple subgoals
- If fail to show proof of one subgoal, reconsider previous subgoal to find alternative solution (backtracking)

Prolog

Backtracking

- Multiple subgoals
- If fail to show proof of one subgoal, reconsider previous subgoal to find alternative solution (backtracking)
- Begin search where previous search left off

Prolog

Backtracking

- Multiple subgoals
- If fail to show proof of one subgoal, reconsider previous subgoal to find alternative solution (backtracking)
- Begin search where previous search left off
- Can take lots of time and space, because may find all possible proofs for every subgoal

Prolog

Backtracking Example:

- Database has:
  male(mike)
  male(bob)
  parent(bob, shelley)

- Goal/query:
  male(X), parent(X, shelley)

Backtracking Example:

- Database has:
  male(mike)
  male(bob)
  parent(bob, shelley)

- Goal/query:
  male(X), parent(X, shelley)

➢ Prolog finds first fact for subgoal male(X) and instantiates X to mike; attempts to prove parent(mike, shelley) but fails

Backtracking Example:

- Database has:
  male(mike)
  male(bob)
  parent(bob, shelley)

- Goal/query:
  male(X), parent(X, shelley)

- Prolog finds first fact for subgoal male(X) and instantiates X to mike; attempts to prove parent(mike, shelley) but fails

- Backtracks to first subgoal male(x); next finds male(bob) such that parent(bob, shelley) is true

Backtracking Example:

- Database has:
  male(mike)
  male(bob)
  parent(bob, shelley)

- Goal/query:
  male(X), parent(X, shelley)

- Prolog finds first fact for subgoal male(X) and instantiates X to mike; attempts to prove parent(mike, shelley) but fails

- Backtracks to first subgoal male(x); next finds male(bob) such that parent(bob, shelley) is true

- To prove goal cannot be satisfied, has to go through all males in database

Prolog

Backtracking Example:

- Database has:
  male(mike)
  male(bob)
  parent(bob, shelley)

- Goal/query:
  male(X), parent(X, shelley)

- Prolog finds first fact for subgoal male(X) and instantiates X to mike; attempts to prove parent(mike, shelley) but fails
- Backtracks to first subgoal male(x); next finds male(bob) such that parent(bob, shelley) is true
- To prove goal cannot be satisfied, has to go through all males in database
- Note: could be more efficient here if reversed order of subgoals

Prolog

Simple arithmetic

- Prolog supports integer variables and arithmetic

Prolog

Simple arithmetic

- Prolog supports integer variables and arithmetic
- Original Prolog had Scheme like + (7 , X)
- Versions today use **is** operator

Prolog

Simple arithmetic

- Versions today use **is** operator

Try in Prolog:

A is 2+3.

10 is 5+5.

10 is 5+2.

A is 5/2.

Prolog

Simple arithmetic

- Versions today use **is** operator

Try in Prolog:

A is 2+3.

10 is 5+5.

10 is 5+2.

A is 5/2.

```
?- A is 2+3.
A = 5.

?- 10 is 5+5.
true.

?- 10 is 5+2.
false.

?- 5/2 is 2.5.
false.

?- A is 5/2.
A = 2.5.
```

Prolog

Simple arithmetic

- All variables on the right must already be instantiated

- A is B/17 + C.
  OK if B and C instantiated

Prolog

Simple arithmetic

- All variables on the right must already be instantiated

- A is B/17 + C.
  OK if B and C instantiated

- Sum is Sum + 1
  OK?

Prolog

Simple arithmetic

- All variables on the right must already be instantiated

- A is B/17 + C.
  OK if B and C instantiated

- Sum is Sum + 1
  No, won’t work! Not like imperative.
  - If Sum is not instantiated, then right side is not proper and cannot assign

Prolog

Simple arithmetic

- All variables on the right must already be instantiated

- A is B/17 + C.
  OK if B and C instantiated

- Sum is Sum + 1
  No, won’t work! Not like imperative.

Prolog

Simple arithmetic

- All variables on the right must already be instantiated

- A is B/17 + C.
  OK if B and C instantiated

- Sum is Sum + 1

➢ If Sum is not instantiated, then right side is not proper and cannot assign

Prolog

Simple arithmetic

- All variables on the right must already be instantiated

- A is B/17 + C.
  OK if B and C instantiated

- Sum is Sum + 1
  - If Sum is not instantiated, then right side is not proper and cannot assign
  - If Sum is instantiated, it is not proper in Prolog to set its left side!

Prolog

Simple arithmetic

- All variables on the right must already be instantiated
- A is B/17 + C.
  OK if B and C instantiated
- Sum is Sum + 1
  - If Sum is not instantiated, then right side is not proper and cannot assign
  - If Sum is instantiated, it is not proper in Prolog to set its left side!
    Not useful or legal in Prolog

Prolog

Prolog example of numeric computation

Example: We know average speed of several automobiles on racetrack and the time on track. We can code relationship speed, time, distance

Prolog

Prolog example of numeric computation

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).

distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

Prolog

Prolog example of numeric computation

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).        Facts
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).

distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

Prolog

Prolog example of numeric computation

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
```

Facts

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

Rule for getting distance:
need to establish the Speed for given X and the Time for given X, and then can set Y to Speed * Time

Prolog

Prolog example of numeric computation

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).

distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

Try the queries:

time(chevy,X).

distance(chevy,X).

distance(X,Y). (with ;)

Prolog

Prolog example of numeric computation

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).

distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

Try the queries:

time(chevy,X).
Returns 21
distance(chevy,X).
Returns 2205 (105*21)
distance(X,Y). (with ;)
Returns all distances

Prolog

How does Prolog keep track of the variables and produce results?

Prolog

How does Prolog keep track of the variables and produce results?

- trace: built in structure displays instantiations of values to variables at each step during attempt to satisfy a goal

Prolog

How does Prolog keep track of the variables and produce results?

- trace: built in structure displays instantiations of values to variables at each step during attempt to satisfy a goal

We will look at this more in our Prolog example in a moment

Prolog

How does Prolog keep track of the variables and produce results?

Prolog's tracing model describes execution as 4 possible events:

1. call, which occurs at beginning of attempt to satisfy a goal

Prolog

How does Prolog keep track of the variables and produce results?

Prolog's tracing model describes execution as 4 possible events:

1. call, which occurs at beginning of attempt to satisfy a goal
2. exit, when goal is satisfied

Prolog

How does Prolog keep track of the variables and produce results?

Prolog's tracing model describes execution as 4 possible events:

1. call, which occurs at beginning of attempt to satisfy a goal
2. exit, when goal is satisfied
3. redo, when backtrack causes attempt to re-satisfy goal

Prolog

How does Prolog keep track of the variables and produce results?

Prolog's tracing model describes execution as 4 possible events:

1. call, which occurs at beginning of attempt to satisfy a goal
2. exit, when goal is satisfied
3. redo, when backtrack causes attempt to re-satisfy goal
4. fail, when goal fails

Prolog

trace for our example

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).

distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

- If distance is thought of as a subprogram, then call and exit can be related to execution models of imperative languages

Prolog

trace for our example

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).

distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

- If distance is thought of as a subprogram, then call and exit can be related to execution models of imperative languages
- Other two events, redo and fail are unique to logical languages

trace for our example

speed.pl

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).

distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

Let’s try it; type in Prolog compiler:

```
trace.
distance(chevy,X).
```

trace for our example

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
   Exit: (7) speed(chevy, 105) ?
   Call: (7) time(chevy, _G1170) ?
   Exit: (7) time(chevy, 21) ?
   Call: (7) _G1097 is 105*21 ?
   Exit: (7) 2205 is 105*21 ?
   Exit: (6) distance(chevy, 2205) ?
X = 2205.
```

Let’s unpack this

trace for our example

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
```

Depth of matching Process. In textbook starts from 1 but not in practice

Internal variable to store instantiated value

Let's unpack this

Prolog

trace for our example

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
```

Depth

Internal variables to store instantiated value

Let's unpack this

Prolog

trace for our example

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
   Exit: (7) speed(chevy, 105) ?
```

Let’s unpack this

Prolog

trace for our example

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
   Exit: (7) speed(chevy, 105) ?
   Call: (7) time(chevy, _G1170) ?
```

Let's unpack this

Prolog

trace for our example

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
   Exit: (7) speed(chevy, 105) ?
   Call: (7) time(chevy, _G1170) ?
   Exit: (7) time(chevy, 21) ?
```

Let's unpack this

Prolog

trace for our example

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
   Exit: (7) speed(chevy, 105) ?
   Call: (7) time(chevy, _G1170) ?
   Exit: (7) time(chevy, 21) ?
   Call: (7) _G1097 is 105*21 ?
```

Let’s unpack this

Prolog

trace for our example

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
   Exit: (7) speed(chevy, 105) ?
   Call: (7) time(chevy, _G1170) ?
   Exit: (7) time(chevy, 21) ?
   Call: (7) _G1097 is 105*21 ?
   Exit: (7) 2205 is 105*21 ?
```

Let’s unpack this

Prolog

trace for our example

```
distance(X,Y) :- speed(X,Speed),
                 time(X,Time),
                 Y is Speed * Time.
```

```
[trace]  ?- distance(chevy,X).
   Call: (6) distance(chevy, _G1097) ?
   Call: (7) speed(chevy, _G1170) ?
   Exit: (7) speed(chevy, 105) ?
   Call: (7) time(chevy, _G1170) ?
   Exit: (7) time(chevy, 21) ?
   Call: (7) _G1097 is 105*21 ?
   Exit: (7) 2205 is 105*21 ?
   Exit: (6) distance(chevy, 2205) ?
X = 2205.
```

Let's unpack this

Prolog

Another example

```
% ['likes.pl'].
% Based on sebesta book
% control d, to exit

likes(jake,chocolate).
likes(jake,apricots).
likes(jake,bananas).
likes(darcie,licorice).
likes(darcie,apricots).
likes(darcie,bananas).
```

In compiler type:

[‘likes.pl’].

likes(jake,X), likes(darcie,X).   Returns?

Another example

In compiler type:
['likes.pl'].
likes(jake,X), likes(darcie,X).

```
Call: (7) likes(jake, _G1097) ? creep
Exit: (7) likes(jake, chocolate) ? creep
Call: (7) likes(darcie, chocolate) ? creep
Fail: (7) likes(darcie, chocolate) ? creep
Redo: (7) likes(jake, _G1097) ? creep
Exit: (7) likes(jake, apricots) ? creep
Call: (7) likes(darcie, apricots) ? creep
Exit: (7) likes(darcie, apricots) ? creep
X = apricots ;
```

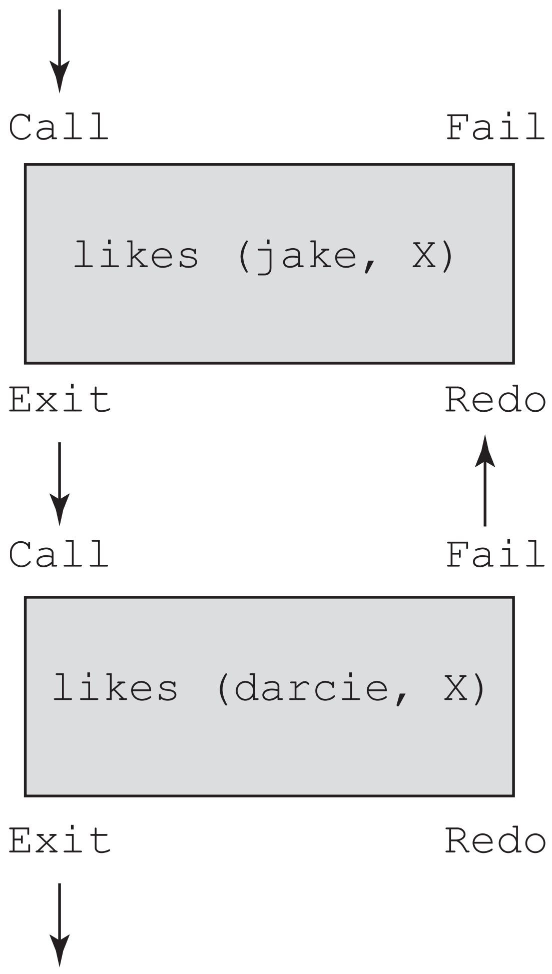
Another example

In compiler type:
['likes.pl'].
likes(jake,X), likes(darcie,X).

(after ;)

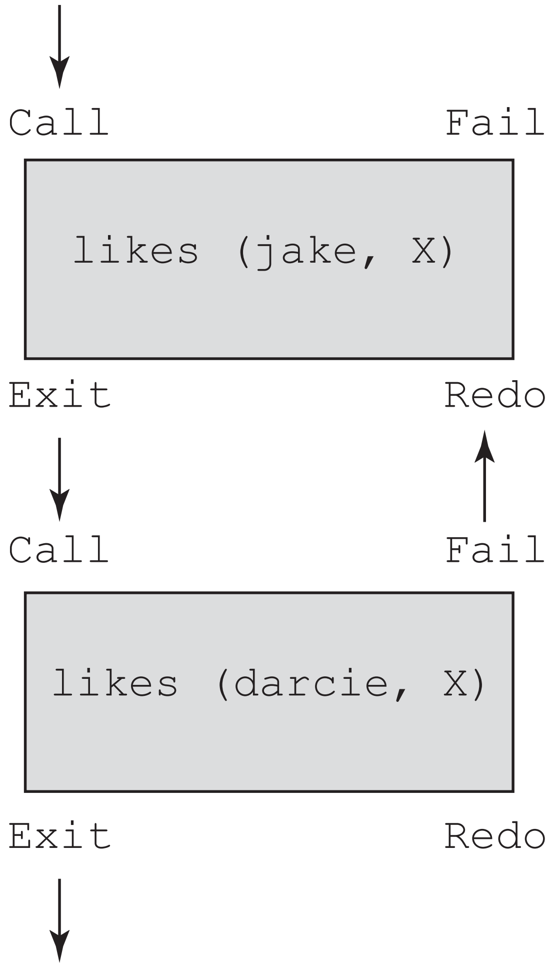
```
X = apricots ;
   Redo: (7) likes(darcie, apricots) ? creep
   Fail: (7) likes(darcie, apricots) ? creep
   Redo: (7) likes(jake, _G1097) ? creep
   Exit: (7) likes(jake, bananas) ? creep
   Call: (7) likes(darcie, bananas) ? creep
   Exit: (7) likes(darcie, bananas) ? creep
X = bananas.
```

Control flow model for
likes(jake,X), likes(darcie,X)



Prolog

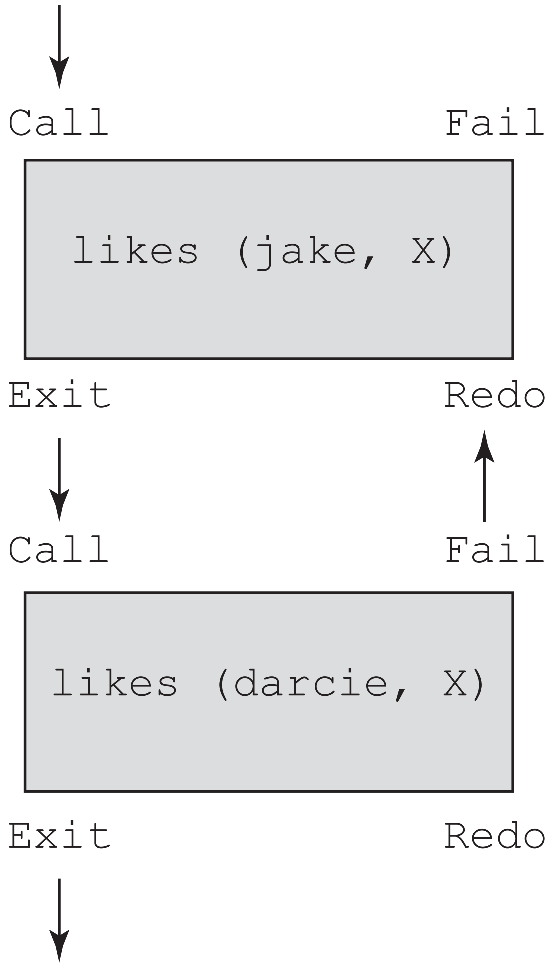
Control flow model for
likes(jake,X), likes(darcie,X)



- Four parts for each subgoal
- Can enter goal through call (forward) or redo (backward)
- Can exit through fail or exit

Prolog

Control flow model for
likes(jake,X), likes(darcie,X)



- Four parts for each subgoal
- Can enter goal through call (forward) or redo (backward)
- Can exit through fail or exit
- Here second subgoal fails the first time, forcing return through redo to first subgoal

Prolog

List structure

- Prolog uses syntax of ML and Haskell to specify lists
- Example: [apple, prune, grape, kumquat]
  [ ] empty list

Prolog

List structure

- Prolog uses syntax of ML and Haskell to specify lists
- Example: [apple, prune, grape, kumquat]
  [ ] empty list
- Prolog also has head and tail:

  [x | y]

  denotes a list with head x and tail y

Prolog

List structure

- Prolog uses syntax of ML and Haskell to specify lists
- Example: [apple, prune, grape, kumquat]
  [ ] empty list
- Prolog also has head and tail:

  [x | y]

  denotes a list with head x and tail y
- **Similar to?**

Prolog

List structure

- Prolog uses syntax of ML and Haskell to specify lists
- Example: [apple, prune, grape, kumquat]
  [ ] empty list
- Prolog also has head and tail:

  [x | y]

  denotes a list with head x and tail y
- Similar to? Most similar to Haskell (x : y) and ML (x :: y) format. Also conceptually related to car, cdr of Scheme.

Prolog

List structure

- Lists can be created by a proposition:
  new_list([apple, prune, grape, kumquat]).

Prolog

List structure

- Lists can be created by a proposition:
  new_list([apple, prune, grape, kumquat]).

- This states that the constant list [apple, prune, grape, kumquat] is a new element of the relation name new_list (a name we just made up).

Prolog

List structure

- Lists can be created by a proposition:
  new_list([apple, prune, grape, kumquat]).

- This states that the constant list [apple, prune, grape, kumquat] is a new element of the relation name new_list (a name we just made up).

- Does a similar thing to male(jake) …
  It states that [apple, prune, grape, kumquat] is a new element of new_list

List structure

- Lists can be created by a proposition:
  new_list([apple, prune, grape, kumquat]).

- This states that the constant list [apple, prune, grape, kumquat] is a new element of the relation name new_list (a name we just made up).

- Does a similar thing to male(jake) …
  It states that [apple, prune, grape, kumquat] is a new element of new_list

- So we can also have a second statement
  new_list([apricot, peach, pear)].

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Run in compiler:

```
new_list(X).
new_list([X|Y]).
```

use ; after entering.

Returns?

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Run in compiler:

```
new_list(X).
?- new_list(X).
X = [apple, prune, grape, kumquot] ;
X = [apricot, peach, pear].
```

Prolog

lists_simple.pl

```
new_list([apple,prune,grape,kumquot]).
new_list([apricot,peach,pear]).
```

Run in compiler:

```
new_list([X|Y]).
```

```
?- new_list([X|Y]).
X = apple,
Y = [prune, grape, kumquot] ;
X = apricot,
Y = [peach, pear].
```

Returns the head and tail of each list!

Prolog

- The | notation can both dismantle and construct lists
- We saw dismantling into a head and tail

Prolog

- The | notation can both dismantle and construct lists
- We saw dismantling into a head and tail
- But we can also construct:
  [pickle, [peanut, prune, popcorn]]

  creates [pickle, peanut, prune, popcorn]

Prolog

- The | notation can both dismantle and construct lists
- We saw dismantling into a head and tail
- But we can also construct:
  [pickle, [peanut, prune, popcorn]]

  creates [pickle, peanut, prune, popcorn]

These are all equivalent!

[apricot, peach, pear | []]
[apricot, peach | [pear]]
[apricot | [peach, pear]]

Prolog

- File lists_simple4.pl

```
% run in compiler:
% new_list(X).
% use ; after entering.

new_list([apricot,peach,pear | []]).
new_list([apricot,peach | [pear]]).
new_list([apricot | [peach,pear]]).
```

Prolog

- File lists_simple4.pl

```
% run in compiler:
% new_list(X).
% use ; after entering.

new_list([apricot,peach,pear | []]).
new_list([apricot,peach | [pear]]).
new_list([apricot | [peach,pear]]).
```

In compiler:

```
?- new_list(X).
X = [apricot, peach, pear] ;
X = [apricot, peach, pear] ;
X = [apricot, peach, pear].
```

Prolog

- File lists_simple4.pl

```
% run in compiler:
% new_list(X).
% use ; after entering.

new_list([apricot,peach,pear | []]).
new_list([apricot,peach | [pear]]).
new_list([apricot | [peach,pear]]).
```

In compiler:

```
?- new_list(X).
X = [apricot, peach, pear] ;
X = [apricot, peach, pear] ;
X = [apricot, peach, pear].
```

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

What does this do??

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

What does this do??

```
?- new_list([apple,prune,grape,kumquot],X,Y).
X = apple,
Y = [prune, grape, kumquot].
```

Returns head and tail

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

What does this do??

```
?- new_list(X,apple,[prune, grape, kumquot]).
X = [apple, prune, grape, kumquot].
```

Constructs list

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

What does this do??

```
?- new_list([apple,prune,grape,kumquot],X,Y).
X = apple,
Y = [prune, grape, kumquot].
```

Returns head and tail

```
?- new_list(X,apple,[prune, grape, kumquot]).
X = [apple, prune, grape, kumquot].
```

Constructs list

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).

?- new_list([apple,prune,grape,kumquot],prune,
[prune, grape, kumquot]).
```

Returns??

Prolog

- File lists_simple2.pl

```
new_list([H|T], H, T).
```

```
?- new_list([apple,prune,grape,kumquot],prune,
[prune, grape, kumquot]).
```

Returns?? false.