Programming in Prolog:  
List

- To form lists:
  - Special constant symbol:  \([\]\)
  - Special function symbol:  \(\text{cons}\)

- Like \textit{Lisp}:

  \[
  \begin{align*}
  \text{Lisp:} & \quad (\text{cons } 'a \ (\text{cons } 'b \ (\text{cons } 'c \ () \ ))) \\
  \text{Prolog:} & \quad \text{cons}( \ a, \ \text{cons}( \ b, \ \text{cons}( \ c, \ [] \ )))
  \end{align*}
  \]

(Lie)
Example: append

- *Prolog* uses *predicates*; not *functions*. Use

\[ \text{append}(X, Y, Z) \]

which holds *iff* Append of X and Y is Z

[not "append(X,Y)" returns "Z"]

- *append* is 3-place predicate:

  1:  \[ \text{append}([], Y, Y). \]
  2:  \[ \text{append}(\text{cons}(E, X), Y, \text{cons}(E, Z)) \]
      \[ :- \text{append}(X,Y,Z). \]

- Notice: If first arg is constant,

  \[ \Rightarrow \text{only 1 head will unify}. \]
Append Goals

1: append( [], Y, Y).
2: append( cons(E, X), Y, cons(E, Z) )
   :- append( X,Y,Z ).

- Goal:

  append( [], cons(a,[]), V1)

  succeeds with

  V1 = cons(a,[])

- Goal:

  append( cons(a, cons(b,[])), cons(c, cons(d,[])), V2)

  succeeds with

  V2 = cons(a,cons(b,cons(c,cons(d,[])))))
Computing append Values

\begin{align*}
\text{append} & \left( \text{cons}(a, \text{cons}(b, [])), \text{cons}(c, \text{cons}(d, [])), V \right) \\
& \quad \text{(2)} \\
& \quad E_1 = a \\
& \quad X_1 = \text{cons}(b, []) \\
& \quad Y_1 = \text{cons}(c, \text{cons}(d, [])) \\
& \quad V = \text{cons}(a, Z_1) \\
\text{append} & \left( \text{cons}(b, []), \text{cons}(c, \text{cons}(d, [])), Z_1 \right) \\
& \quad \text{success} \\
\text{append} & \left( [], \text{cons}(c, \text{cons}(d, [])), Z_2 \right) \\
& \quad \text{(1)} \\
& \quad Y_2 = \text{cons}(c, \text{cons}(d, [])) \\
& \quad Z_2 = \text{cons}(c, \text{cons}(d, [])) \\
V & = \text{cons}(a, Z_1) = \text{cons}(a, \text{cons}(b, Z_2)) \\
& \quad = \text{cons}(a, \text{cons}(b, \text{cons}(c, \text{cons}(d, []))))
\end{align*}
Prolog's List Shorthand

- Prolog's convention:
  
  \[ [s_1, \ldots, s_n \mid t] \text{ is abbreviation for } \]
  \[ \text{cons}(s_1, \text{cons}(s_2, \ldots \text{cons}(s_n, t) \ldots)) \]
  
  \[ [s_1, \ldots, s_n] \text{ is abbreviation for } \]
  \[ \text{cons}(s_1, \text{cons}(s_2, \ldots \text{cons}(s_n, []) \ldots)) \]
  
  (ie, for \([s_1, \ldots, s_n \mid [] \])

- Like Lisp:
  
  \((s_1 \cdots s_n \cdot s)\) is evaluation of
  
  \((\text{cons} \ 's_1 \ (\text{cons} \ 's_2 \ \ldots \ (\text{cons} \ 's_n \ 's) \ \ldots))\)
  
  \((s_1 \cdots s_n)\) is evaluation of
  
  \((\text{cons} \ 's_1 \ (\text{cons} \ 's_2 \ \ldots \ (\text{cons} \ 's_n \ ()) \ \ldots))\)
  
  (ie, for \((s_1 \cdots s_n \cdot ())\))

- Lisp \(\rightarrow\) Prolog:
  
  - Change "( )" to "[ ]"
  - Add ",," between args
  
  "(a + b * c)" \(\rightarrow\) "[a, +, b, *, c]"
Alternate Description of append

Can use abbreviation:

?- [user].
append([], Y, Y).
append([E | X], Y, [E | Z]) :- append(X, Y, Z).
:D user con...
yes
?- append([a, b], [c, d], V).
V = [a, b, c, d] _
yes
?-

This is SAME definition!
“Expands” into same def’n shown earlier!
Computing `append` Values

\[
\text{append( } [a,b], [c,d], V \text{ )}
\]

(2)
\[
\begin{align*}
E_1 &= a \\
X_1 &= [b] \\
Y_1 &= [c,d] \\
V &= [a \mid Z_1]
\end{align*}
\]

\[
\text{append( } [b], [c,d], Z_1 \text{ )}
\]

(2)
\[
\begin{align*}
E_2 &= b \\
X_2 &= [] \\
Y_2 &= [c,d] \\
Z_1 &= [b \mid Z_2]
\end{align*}
\]

\[
\text{append( } [], [c,d], Z_2 \text{ )}
\]

(1)
\[
\begin{align*}
Y_2 &= [c,d] \\
Z_2 &= [c,d]
\end{align*}
\]

\textit{success}

\[\Rightarrow \quad V = [a \mid Z_1] = [a \mid [b \mid Z_2]] = [a,b,c,d]\]
Verifying Append Values

Does “append( [a,b], [c] )” = “[a,b,c,d]”?

append( [a,b], [c], [a,b,c,d] )

(2)
E₁ = a
X₁ = [b]
Y₁ = [c]
Z₁ = [b,c,d]

append( [b], [c], [b,c,d] )

(2)
E₂ = b
X₂ = []
Y₂ = [c]
Z₂ = [c,d]

append( [], [c], [c,d] )

No possible unification!

X
Using append to extract SubList

Find $W$ s.t. $append([a], W) = [a,b,c]$

\[
 append( [a], W, [a,b,c] )
\]

(2)
- $E_1 = a$
- $X_1 = []$
- $Y_1 = W$
- $Z_1 = [b,c]$

\[
 append( [], W, [b,c] )
\]

(1)
- $W = [b,c]$

\[
 \text{success}
\]

$\Rightarrow W = Y_1 = [b,c]$. 

Similarly,

$append(D, [c,d], [a,b,c,c,d])$

succeeds with $D = [a,b,c]$. 

Other uses of append

- append([W, b, c], [d, e], [a, b, c, d, e])
  succeeds with  \( W = a \)

- append([a, b, c], [d, e], [a, b, W, d, e])
  succeeds with  \( W = c \)

- append([a, b, c], [d, e], [a, b \mid W])
  succeeds with  \( W = [c, d, e] \)

- append([a, b, c], [d \mid W], [a, b, c, d, e, f])
  succeeds with  \( W = [e, f] \)

- append([a, X, c], [Y, e], [a, b, Z, d, e])
  succeeds with  \( X = b, Y = d, Z = c \).

- What about  \( \text{append}(U, V, [a, b]) \)?
append( U, V, [a,b] )

(1) 
U = []
V = [a,b]

success
U = []
V = [a,b]

append( U, V, [a,b] )

(2) 
E_1 = a
U = [a | X_1]
V = Y_1
Z_1 = [b]

append( X_1, Y_1, [b] )

(2) 
E_2 = b
X_1 = [b | X_2]
Y_1 = Y_2
Z_2 = []

append(X_2, Y_2, [])

(1) 
X_2 = []
Y_2 = []

success
U = [a]
V = [b]

(1) 
X_1 = []
Y_1 = [b]

success
U = [a]
V = [b]

(1) 
X_2 = []
Y_2 = []

success
U = [a,b]
V = []
append and Variables

- **append(A, B, [a,b,c,d,e])** succeeds with
  
  \[
  \begin{align*}
  A &= [] & B &= [a,b,c,d,e] \\
  A &= [a] & B &= [b,c,d,e] \\
  A &= [a,b] & B &= [c,d,e] \\
  A &= [a,b,c] & B &= [d,e] \\
  A &= [a,b,c,d] & B &= [e] \\
  A &= [a,b,c,d,e] & B &= []
  \end{align*}
  \]
  
- What about **append(A, B, C)**?
  
  succeeds with \( A=[] \) \( B= \_1 \) \( C= \_1 \)
  
  then \( A=\_8 \) \( B= \_1 \) \( C=[\_8 \mid \_1] \)
  
  then \( A=\_8,\_15 \) \( B= \_1 \) \( C=[\_8,\_15 \mid \_1] \)

  ...forever...
append( A, B, C )

(1)
A = []
B = _1
C = _1

success
A = []
B = _1
C = _1

append( X_1, Y_1, Z_1 )

(2)
A = [E_1 | X_1]
B = Y_1
C = [E_1 | Z_1]

append( X_2, Y_2, Z_2 )

(1)
X_1 = []
Y_1 = _1
Z_1 = _1

success
A = [ _8 ]
B = _1
C = [ _8 | _1]

etc. (forever)

(1)
X_2 = []
Y_2 = _15
Z_2 = _15

success
A = [ _8, _15 ]
B = _1
C = [ _8, _15 | _1]
append Goals

| ?- [user].
| append( [], Y, Y ).

?- user con...
yes

?- append( [a, b, c], [d, e], V ).
V = [a, b, c, d, e] _

yes

?- append( [a, X, c], [Y, e], [a, b, Z, d, e] ).
X = b
Y = d
Z = c ;

no % Says "no" as user asked for 2nd answer.

?- append( U, V, [a,b,c] ).
U = []
V = [a,b,c] _

U = [a]
V = [b,c] _

U = [a,b]
V = [c] _

U = [a,b,c]
V = [] _

no

.
Other Uses of append

- last element of a list
  \[ \text{last( [a,b,c], c )} \]
  \[ \text{last( L, E ) :- append( Y, [E], L )} \]

- member of a list
  \[ \text{member( c, [a,b,c] )} \]
  \[ \text{member( E, L ) :- append( Y, [E | Z], L )} \]

- prefix at beginning of a list
  \[ \text{prefix( [a,b], [a,b,c,d] )} \]
  \[ \text{prefix( L1, L3 ) :- append( L1, L2, L3 )} \]

- suffix at end of list
  \[ \text{suffix( [c,d], [a,b,c,d] )} \]
  \[ \text{suffix( L2, L3 ) :- append( L1, L2, L3 )} \]

- middle of a list
  \[ \text{middle( [a,b,c,d,e,f], [a,b], [e,f], [c,d] )} \]
  \[ \text{middle( L, L1, L3, L2 ) :- append( L1, L2, L12 ), append( L12, L3, L )}. \]
How to Solve Problems (Prolog)

• I could solve the BIG problem if...
  I could solve some specific subproblem

Eg: I could solve

  mortal(socrates)
if I knew that

  man(socrates)

  Note just sufficiency – not necessity!

So

  mortal(socrates) :- man(socrates).

• Generality: In fact...

  mortal(X) :- man(X).
How to Solve Problems (con’t)

Eg: I could solve
    append( [E | X], Y, ??)
    if I could solve
    append( X, Y, Z)

    Then answer would be
    ?? = [E | Z]

    So


- One more thing...
  Some things are just true — “base case”
  append( [], Y, Y ).
Other Examples of List Programming

• Equality of Lists
  
  equal(X,X).

  (Easy: unification does all of the work!)

  Note: built into Prolog as "="

  \[=\!(t_1,\ t_2)\] succeeds iff \(t_1\) and \(t_2\) unifiable.

• List membership:

  member(X, [X | L]). \hspace{1cm} (5)

  member(X, [Y | L]) :~ member(X,L). \hspace{1cm} (6)

  Can be used to generate elements of list:

  \[\text{member}(X, [a,b,c])\] succeeds with \(X=a\)
  then, if required, \(X=b\)
  then, if required, \(X=c\).

  Useful for trying fixed set of possibilities.
**Variants of member**

- **Alternative Definition #1**

  member(X, [X | _]). \hspace{1cm} (1)
  member(X, [_ | L]) :- member(X,L). \hspace{0.5cm} (2)

  “_” is variable, unifies with ANYTHING
  Don't care about its value
  Each occurance can be different.

- **Alternative Definition #2**

  nmember(X, [_ | L]) :- nmember(X,L). \hspace{0.5cm} (1)
  nmember(X, [X | _]). \hspace{1cm} (2)

(Same information, different order.)

Observe:
- `member(a,[a,b,c,d,e])` requires 1 step
- `nmember(a,[a,b,c,d,e])` requires 6 steps
- `member(c,[a,b,c,d,e])` requires 3 steps
- `nmember(c,[a,b,c,d,e])` requires 6 steps
Length of Proofs — member

\[ KB_1 = \{ \text{member}(X, [X \mid \_]). \quad (1) \}
\]

\[ \text{member}(X, [\_ \mid L]) :\text{-} \text{member}(X, L). \quad (2) \}

member( c, [a,b,c,d,e] )

\[ \text{member}( c, [\_ \mid a, L_1 / [b,c,d,e] ) \quad (2) \]

\[ \{ x_1 / c, \_ / a, L_1 / [b,c,d,e] \} \]

member( c, [b,c,d,e] )

\[ \text{member}( c, [\_ \mid b, L_2 / [c,d,e] ) \quad (2) \]

\[ \{ x_2 / c, \_ / b, L_2 / [c,d,e] \} \]

member( c, [c,d,e] )

\[ \text{member}( c, [\_ / [d,e] \} \quad (1) \]

\[ \{ x_3 / c, \_ / [d,e] \} \]

success
Length of Proofs — nmember

\[ KB_2 = \begin{cases} 
\text{nmember}(X, [\_ | L]) : - \text{nmember}(X, L). & (1') \\
\text{nmember}(X, [X | \_]). & (2') 
\end{cases} \]

\textbf{nmember}(c, [a,b,c,d,e])

\begin{align*}
\{X_1 / c, \_/ a, L_1 / [b,c,d,e]\} &
\rightarrow \text{nmember}(c, [b,c,d,e]) \\
\{X_2 / c, \_/ b, L_2 / [c,d,e]\} &
\rightarrow \text{nmember}(c, [c,d,e]) \\
\{X_3 / c, \_/ c, L_3 / [d,e]\} &
\rightarrow \text{nmember}(c, [d,e]) \\
\{X_4 / c, \_/ d, L_4 / [e]\} &
\rightarrow \text{nmember}(c, [\_]) \\
\{X_5 / c, \_/ e, L_5 / [\_]\} &
\rightarrow \text{nmember}(c, [\_]) \\
& \text{success} \\
& \text{FAIL} \\
\end{align*}
**Big Example: Sorting**

- \( \text{sort}(X,Y) \) holds iff
  - \( Y \) is a permutation of \( X \)
  - \( Y \) is in ascending order

- Perhaps:
  
  \[
  \text{sort}(X,Y) :- \text{permute}(X,Y), \text{ordered}(Y).
  \]

- **Terrible:** \( n! \) permutations of \( n \)-element \( X \!\)

\[
\text{sort}( [3,1,2], [1,2,3] ) \quad \text{is ok, but}\n\]
\[
\text{sort}( [3,1,2], Y ) \quad \sim \quad \text{permute}( [3,1,2], Y ), \text{ordered}(Y) \]
\[
\sim Y / [3,1,2]
\]
\[
Y / [3,2,1]
\]
\[
Y / [2,3,1]
\]
\[
Y / [2,1,3]
\]
\[
Y / [1,3,2]
\]
\[
Y / [1,2,3]
\]
Quick Sort

Algorithm

Choose item in list as “pivot”.
Put all items < pivot into L1.
Put all items > pivot into L2.
Sort L1, Sort L2.
Append the results (L1', pivot, L2')

In Prolog:

```prolog
qsort([], [])  % Base case
qsort([Pivot|Rest], Sorted) :-
    part(Pivot, Rest, L1, L2),
    qsort(L1, Lessers), qsort(L2, Greaters),
    append(Lessers, [Pivot|Greaters], Sorted).

% Now to split the list, into elements < and > the Pivot value
part(Pivot, [], [], [])
part(Pivot, [Item|Rest], [Item|L1], L2) :-
    lessThan(Item, Pivot), part(Pivot, Rest, L1, L2).
part(Pivot, [Item|Rest], L1, [Item|L2]) :-
    notLessThan(Item, Pivot), part(Pivot, Rest, L1, L2).
```