Programming in Prolog: List

- To form lists:
  - Special constant symbol: `[]`
  - Special function symbol: `cons`

- Like *Lisp*:

```
Lisp:     (cons 'a  (cons 'b  (cons 'c  () )))
Prolog:    cons( a,  cons( b,  cons( c, [] )))
```

(Lie)

Example: `append`

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

  $$\texttt{append(X,Y,Z)}$$

  which holds *iff* `Append` of `X` and `Y` is `Z`
  [not "`append(X,Y)`" returns "`Z`"]

- `append` is 3-place predicate:

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

- Notice: If first arg is constant,
  $\Rightarrow$ 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

append(`cons(a, cons(b,[])), cons(c, cons(d,[])),` **V)**

(2)

E_1 = a
X_1 = cons(b, [])
Y_1 = cons(c, cons(d,[]))
V = cons(a, Z_1)

append(`cons(b,[]), cons(c, cons(d,[])),` **Z_1)**

(2)

E_2 = b
X_2 = []
Y_2 = cons(c, cons(d,[]))
Z_1 = cons(b, Z_2)

append(`[], cons(c, cons(d,[])),` **Z_2)**

(1)

Y_2 = cons(c, cons(d,[]))
Z_2 = cons(c, cons(d,[]))

success

$\Rightarrow$ V = cons(a, Z_1) = cons(a, cons(b, Z_2))

= cons(a, cons(b, cons(c, cons(d, []))))

Prolog's List Shorthand

- *Prolog*'s convention:

  $\texttt{[s}_1\texttt{, } \cdots\texttt{, s}_n \texttt{ | t]}$ is abbreviation for

  $\texttt{cons(s}_1\texttt{, cons(s}_2\texttt{, } \cdots \texttt{ cons(s}_n\texttt{, t) } \cdots\texttt{))}$

  $\texttt{[s}_1\texttt{, } \cdots\texttt{, s}_n\texttt{]}$ is abbreviation for

  $\texttt{cons(s}_1\texttt{, cons(s}_2\texttt{, } \cdots \texttt{ cons(s}_n\texttt{, []) } \cdots\texttt{))}$

  (ie, for $\texttt{[s}_1\texttt{, } \cdots\texttt{, s}_n \texttt{ | []]}$)

- Like *Lisp*:

  $\texttt{(s}_1 \cdots \texttt{s}_n \texttt{ . s)}$ is evaluation of

  $\texttt{(cons 's}_1 \texttt{ (cons 's}_2 \cdots \texttt{ (cons 's}_n \texttt{ 's) } \cdots\texttt{))}$

  $\texttt{(s}_1 \cdots \texttt{s}_n\texttt{)}$ is evaluation of

  $\texttt{(cons 's}_1 \texttt{ (cons 's}_2 \cdots \texttt{ (cons 's}_n \texttt{ ()) } \cdots\texttt{))}$

  (ie, for $\texttt{(s}_1 \cdots \texttt{s}_n \texttt{ . ())}$)

- *Lisp* $\leadsto$ *Prolog*:
  - Change "( )" to "[ ]"
  - Add "," between args

  "(a + b * c)" $\leadsto$ "[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

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

(2)

E_1 = a
X_1 = [b]
Y_1 = [c,d]
V = [a | Z_1]

```
append( [b], [c,d], Z1 )
```

(2)

E_2 = b
X_2 = []
Y_2 = [c,d]
Z_1 = [b | Z_2]

```
append( [], [c,d], Z2)
```

(1)

Y_2 = [c,d]
Z_2 = [c,d]

```
success
```

⇒ V = [a | Z_1] = [a | [b | Z_2]]

= [a | [b | [c,d]]] = [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)
        |     E1 = a
        |     X1 = []
        |     Y1 = W
        |     Z1 = [b,c]
        |
append( [], W, [b,c] )
        |  (1)
        |     W = [b,c]
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`

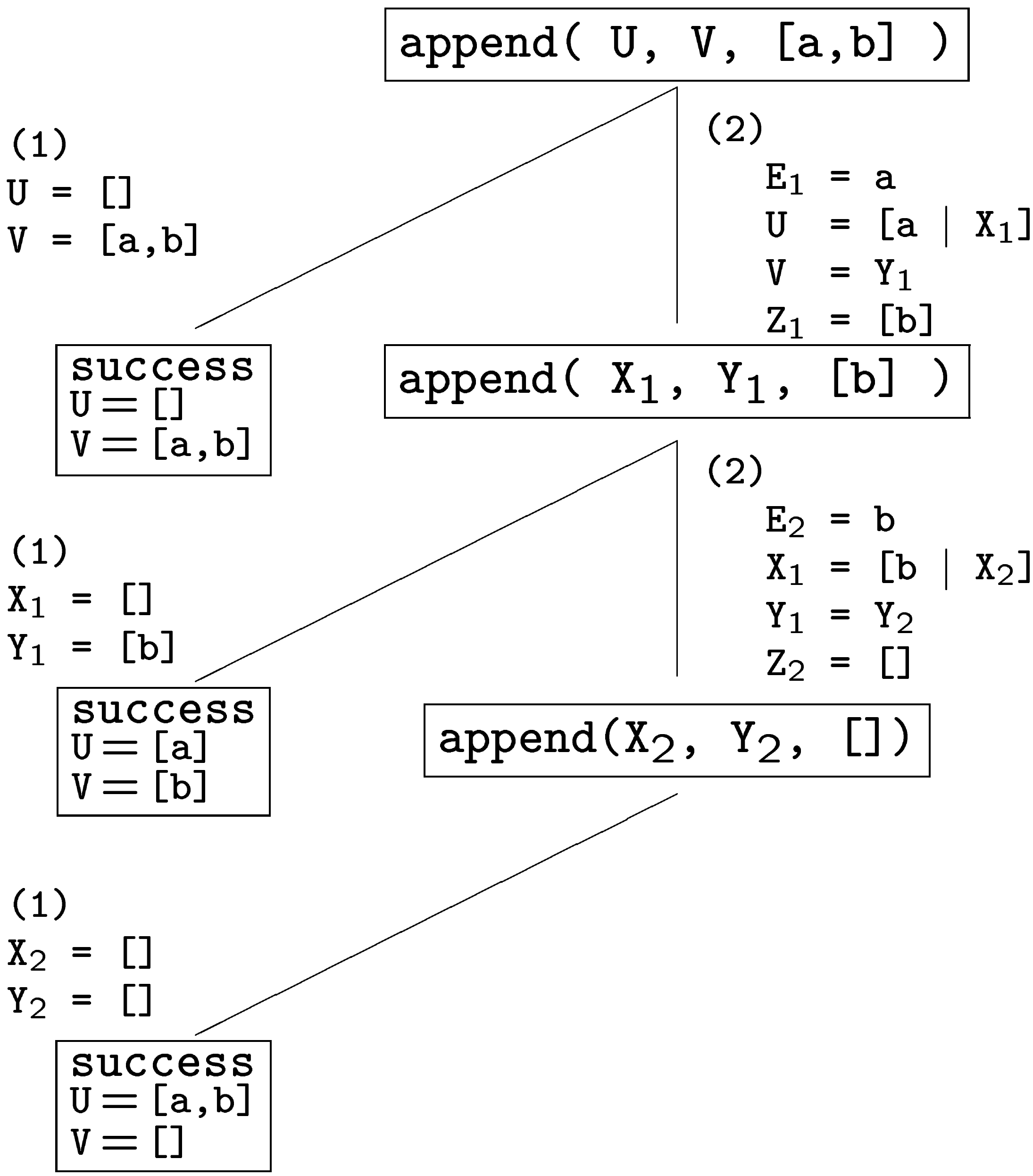
- `append([a,b,c], [d,e], [a,b | W])`
  succeeds with `W = [c,d,e]`

- `append([a,b,c], [d | 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 `append(U, V, [a,b])` ?

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



append(U, V, [a,b])
(1)
U = []
V = [a,b]
(2)
E_1 = a
U = [a | X_1]
V = Y_1
Z_1 = [b]
success
U = []
V = [a,b]
append(X_1, Y_1, [b])
(1)
X_1 = []
Y_1 = [b]
(2)
E_2 = b
X_1 = [b | X_2]
Y_1 = Y_2
Z_2 = []
success
U = [a]
V = [b]
append(X_2, Y_2, [])
(1)
X_2 = []
Y_2 = []
success
U = [a,b]
V = []

`append` and Variables

- `append(A, B, [a,b,c,d,e])` succeeds with

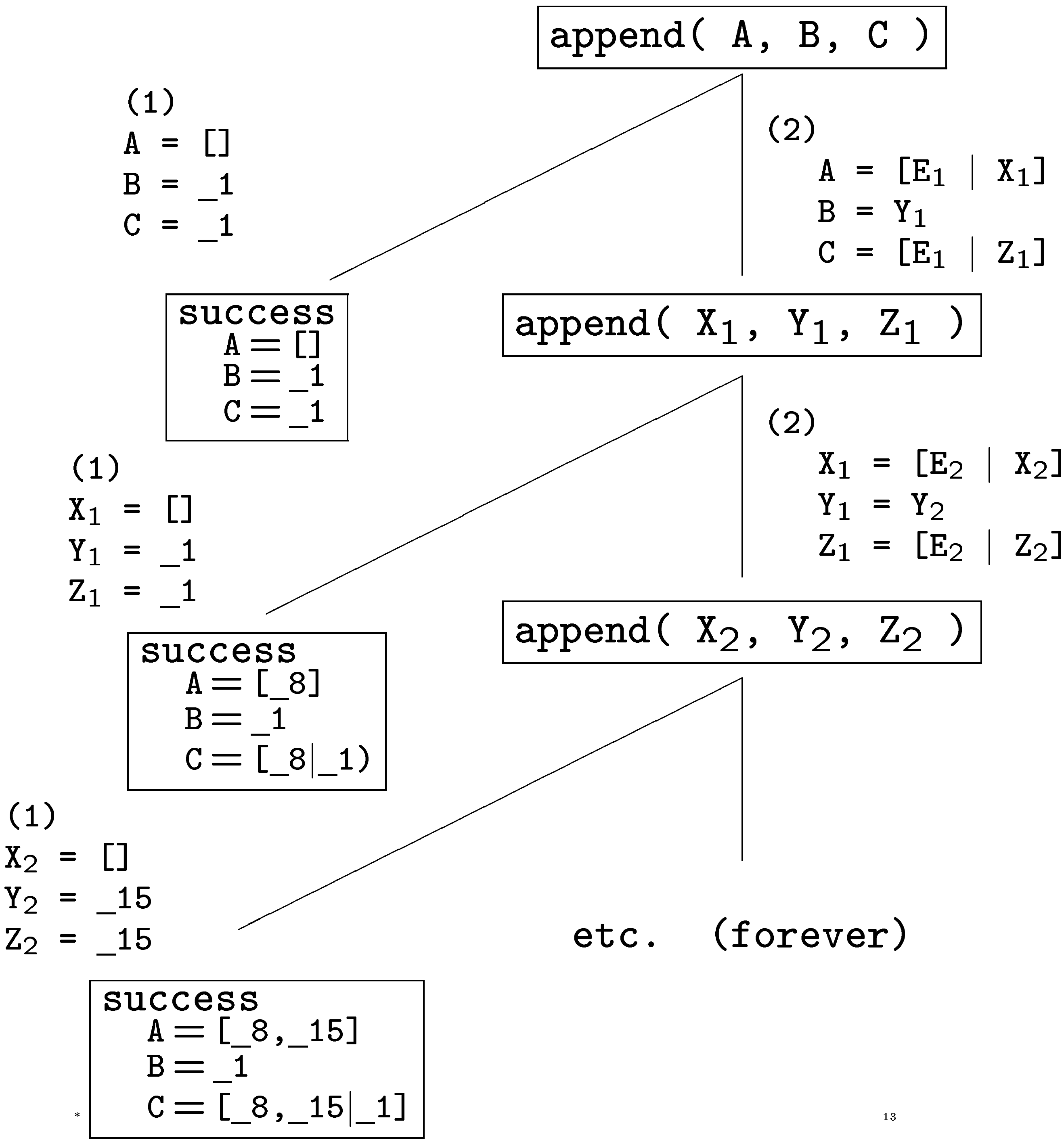
A = []	B = [a,b,c,d,e]
A = [a]	B = [b,c,d,e]
A = [a,b]	B = [c,d,e]
A = [a,b,c]	B = [d,e]
A = [a,b,c,d]	B = [e]
A = [a,b,c,d,e]	B = []

- What about `append(A, B, C)` ?

succeeds with	A=[]	B=_1	C=_1
then	A=[_8]	B=_1	C=[_8 | _1]
then	A=[_8,_15]	B=_1	C=[_8,_15| _1]

. . . forever . . .

append(A, B, C) ?


append(A, B, C)
(1)
A = []
B = _1
C = _1
(2)
A = [E1 | X1]
B = Y1
C = [E1 | Z1]
success
A = []
B = _1
C = _1
append(X1, Y1, Z1)
(1)
X1 = []
Y1 = _1
Z1 = _1
(2)
X1 = [E2 | X2]
Y1 = Y2
Z1 = [E2 | Z2]
success
A = [_8]
B = _1
C = [_8|_1)
append(X2, Y2, Z2)
(1)
X2 = []
Y2 = _15
Z2 = _15
etc. (forever)
success
A = [_8,_15]
B = _1
C = [_8,_15|_1]

append Goals

```
| ?- [user].

| append( [], Y, Y ).

| append( [E | X], Y, [E | Z] ) :- append( X,Y,Z ).

| ^D 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

```
    last( [a,b,c], c )
last( L, E ) :- append( Y, [E], L )
```

- `member` of a list

```
    member( c, [a,b,c] )
member( E, L ) :- append( Y, [E | Z], L )
```

- `prefix` at beginning of a list

```
    prefix( [a,b], [a,b,c,d] )
prefix( L1, L3 ) :- append( L1, L2, L3 )
```

- `suffix` at end of list

```
    suffix( [c,d], [a,b,c,d] )
suffix( L2, L3 ) :- append( L1, L2, L3 )
```

- `middle` of a list

```
    middle( [a,b,c,d,e,f], [a,b], [e,f], [c,d] )
 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

```
append( [E|X], Y, [E|Z] ) :- append( X, Y, Z ).
```

- 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 "="

`=(t1, t2)` succeeds iff t_1 and t_2 unifiable.

- List membership:

member(X, [X | L]). *(5)*
member(X, [Y | L]) :- member(X,L). *(6)*

Can be used to *generate* elements of list:

`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 | _]). *(1)*
  member(X, [_ |L]) :- member(X,L). *(2)*

  "_" is variable, unifies with ANYTHING

  Don't care about its value

  Each occurrance can be different.

- Alternative Definition #2

  nmember(X, [_ |L]) :- nmember(X,L). *(1)*
  nmember(X, [X | _]). *(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 =

```
member(X, [X | _]).                     (1)
member(X, [_ |L]) :- member(X,L).       (2)
```

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

(2)
{ X_1/c, _/a, L_1/[b,c,d,e] }

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

(2)
{ X_2/c, _/b, L_2/[c,d,e] }

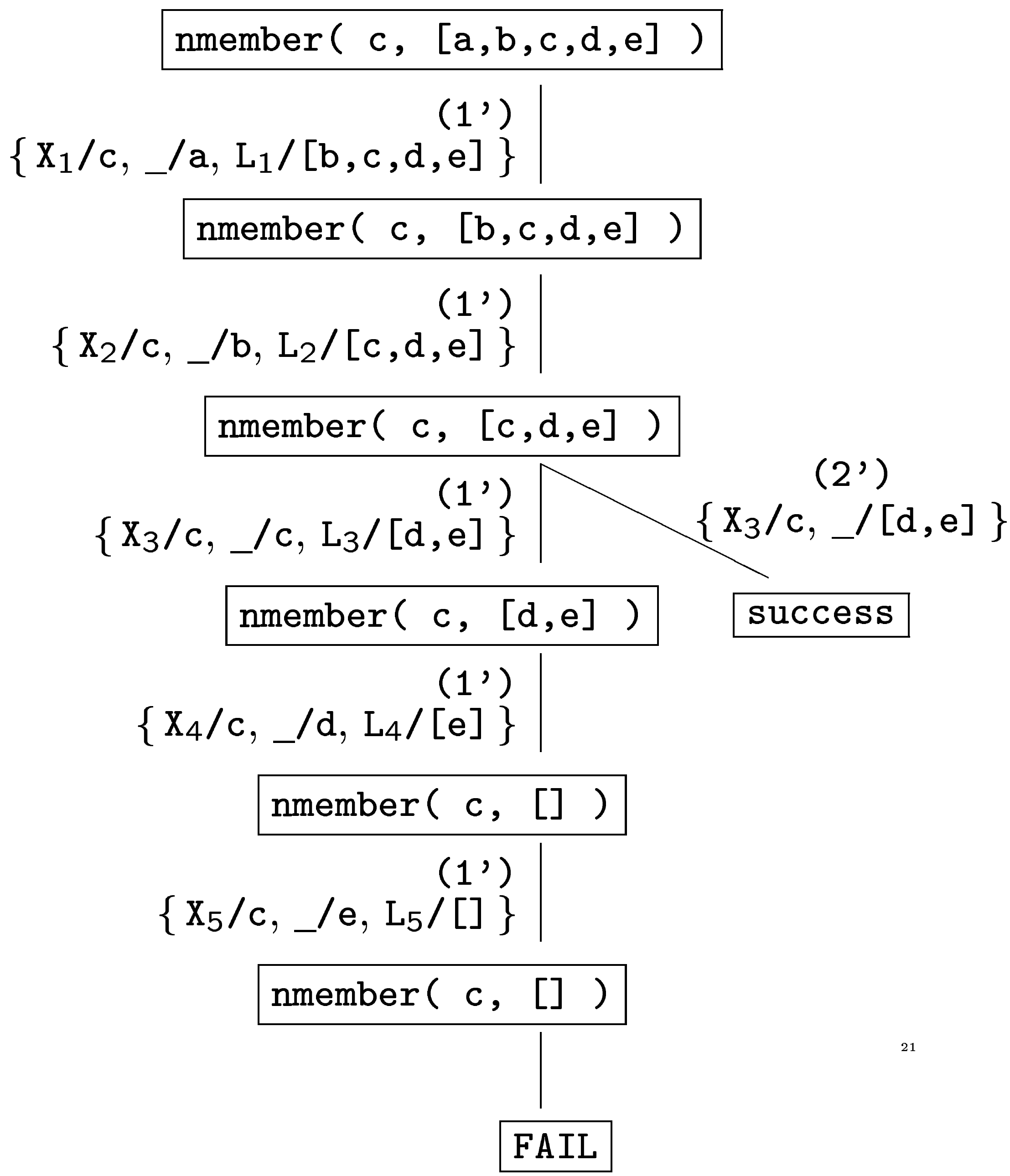
member(c, [c,d,e])

(1)
{ X_3/c, _/[d,e] }

success

Length of Proofs – nmember

$$KB_2 = \left\{ \begin{array}{ll} \texttt{nmember(X, [\_ |L]) :- nmember(X,L).} & \texttt{(1')} \\ \texttt{nmember(X, [X | \_]).} & \texttt{(2')} \end{array} \right\}$$



Big Example: Sorting

- `sort(X,Y)` holds iff

  `Y` is a permuation of `X`

  `Y` is in ascending order

- Perhaps:

```
sort(X,Y) :- permute(X,Y), ordered(Y).
```

- *Terrible:* $n!$ *permutations of* n*-element* `X`*!*

`sort( [3,1,2], [1,2,3] )` is ok, but

`sort( [3,1,2], Y )`

$\leadsto$ `permute( [3,1,2], Y ), ordered(Y)`

$\leadsto$ `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*:

```
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).
```