3 How Prolog Works

1. Resolution
2. Backtracking
3. Generate and Test
4. The Debugger

3.1 Resolution

Prolog's execution mechanism is based on **resolution**

Much more detail on resolution later; for now a very pragmatic view

Actual execution strategy is more efficient than this presentation — more like the way conventional languages are implemented

This description leaves off how unification works

This is only a rough sketch

3.1.1 The Basic Algorithm

1. Start with a query $g_1, g_2, g_3, \ldots$ and empty stack
2. If the query is empty, terminate *succeeding*; otherwise, choose the first goal from query
3. If no program clause matches the chosen goal, then:
4. If the stack is empty, terminate *failing*; otherwise pop the query and clause from the stack and return to 3
5. Otherwise: choose first clause whose head matches the chosen goal; push query and next clause on a stack
6. Replace first goal with the body of the chosen clause at the front of the query and return to 2

3.1.2 Example (without backracking)

```
add(0, Y, Y).
add(s(X), Y, s(Z)) :-
        add(X, Y, Z).
```

Initial query: `add(s(s(0)), s(s(s(0))), A).`

Clause 1 does not match; clause 2 matches with

`X = s(0),    Y = s(s(s(0))),    A = s(Z)`

New query: `add(s(0), s(s(s(0))), Z)`

Clause 1 does not match; clause 2 matches with

`X1 = 0,    Y1 = s(s(s(0))),    Z = s(Z1)`

(variable names changed)

New query: `add(0, s(s(s(0))), Z1)`

Clause 1 matches with `Z1 = Y2 = s(s(s(0)))`

New query is empty: success, leaving

`A = s(Z) = s(s(Z1)) = s(s(s(s(s(0)))))`

3.2 Backtracking

Nondeterminism describes a computation that may have more than one result, for example

```
?- parent(diana, X).
```

Support for nondeterminism is an important feature of Prolog

Prolog handles nondeterminism by **backtracking** — undoing all work done since a tentative choice was made so an alternative choice can be tried

Backtracking is performed in steps 3 and 4, when multiple clauses matched a selected goal, and later a goal fails

An entry pushed on the stack in the resolution algorithm is called a **choicepoint**

3.2.1 Backtracking Example

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

Initial query: `member(Z, [a,b]), member(Z, [b,c])`

Both clauses match first goal; choose first and **push a choicepoint**

```
Z = X = a
```

New query: `member(a, [b,c])`

Clause 1 does not match; clause 2 matches with

```
X1 = a,    L1 = [c]
```

Backtracking Example (2)

New query: $\boxed{\texttt{member(a, [c])}}$

Clause 1 does not match; clause 2 matches with

```
X2 = a,    L2 = []
```

New query: $\boxed{\texttt{member(a, [])}}$

Neither clause matches: failure

Backtracking Example (3)

We are not done; we pop the choicepoint off the stack

This returns us to state at time the choicepoint was pushed, but now we go on to the next clause

New query: `member(Z, [a,b]), member(Z, [b,c])`

Now choose second clause, with

```
Z = X,    L = [b]
```

New query: `member(Z, [b]), member(Z, [b,c])`

Both clauses match first goal; choose first and **push a choicepoint**

```
Z = X = b
```

Backtracking Example (4)

New query: `member(b, [b,c])`

Both clauses match first goal; choose first and **push a choicepoint**

```
X2 = b
```

New query is empty: success, leaving

```
Z = b
```

Prolog prints this result; if we hit `;` asking for more solutions, this forces an artificial failure, causing Prolog to backtrack, looking for more solutions

Backtracking Example (5)

New query: `member(b, [b,c])`

Clause 2 matches with `X2 = b,    L2 = [c]`

New query: `member(b, [c])`

Clause 2 matches with `X3 = b,    L3 = []`

New query: `member(b, [])`

Neither clause matches: pop last choicepoint

New query: `member(Z, [b]), member(Z, [b,c])`

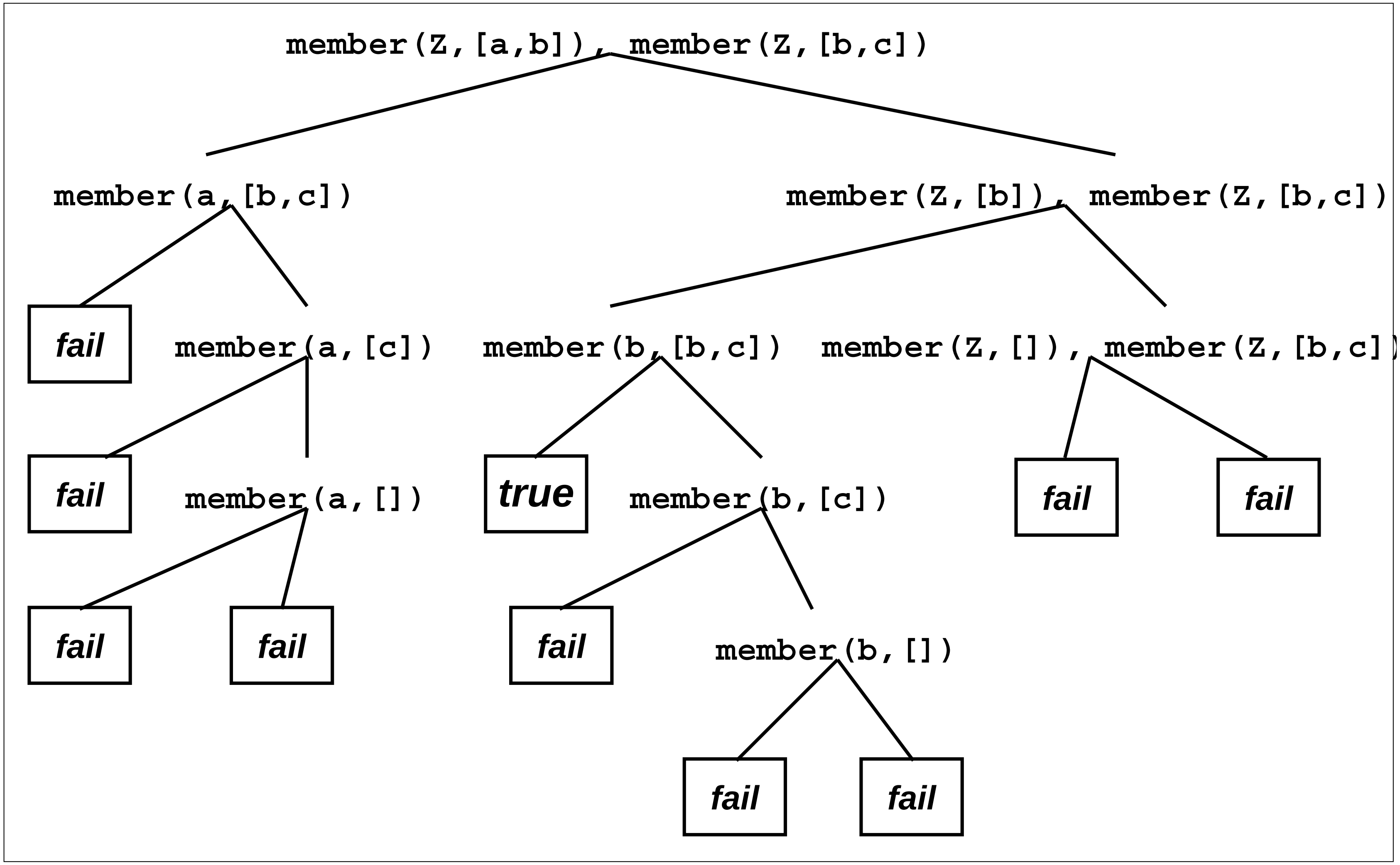
Second clause matches, leaving

```
X3 = Z,    L3 = []
```

New query: `member(Z, []), member(Z, [b,c])`

Neither clause matches: final failure

3.2.2 Example: Search Tree



Exercise: Motherhood Again

Draw a search tree for the following program and query

```
parent(charles, william). parent(charles, harry).
parent(diana, william).   parent(diana, harry).
female(elizabeth).        female(diana).
?- parent(X,Y), female(X).
```

3.3 Generate and Test

This example shows that the two calls to `member` behave very differently:

First call `member(Z, [a,b])` successively **generates** elements of the list `[a,b]`

Second call `member(Z, [b,c])` **tests** the solutions generated by the first call

This is because when first call is made, `Z` is unbound, but when second call is made, `Z` is bound

> **generate and test** is a simple but powerful technique for solving compound constraints

> The bound/unbound state of the arguments of a predicate invocation is called its **mode**

3.3.1 Reverse

A list can be reversed by appending its first element to the reverse of the remaining elements:

```
rev([], []).               % the reverse of [] is []
rev([A|BC], R) :-          % the reverse of a list [A] ++ BC
  rev(BC, CB),             %     is the reverse of BC
  append(CB, [A], R).      %     with A added on the end
```

Unfortunately, this definition is not as flexible as we would like:

```
?- rev([a,b,c], R).
R = [c, b, a] ;
No
```

```
?- rev(R, [a,b,c]).
R = [c, b, a] ;
Action (h for help) ? abort
% Execution Aborted
```

3.3.2 Reverse Goes Wrong

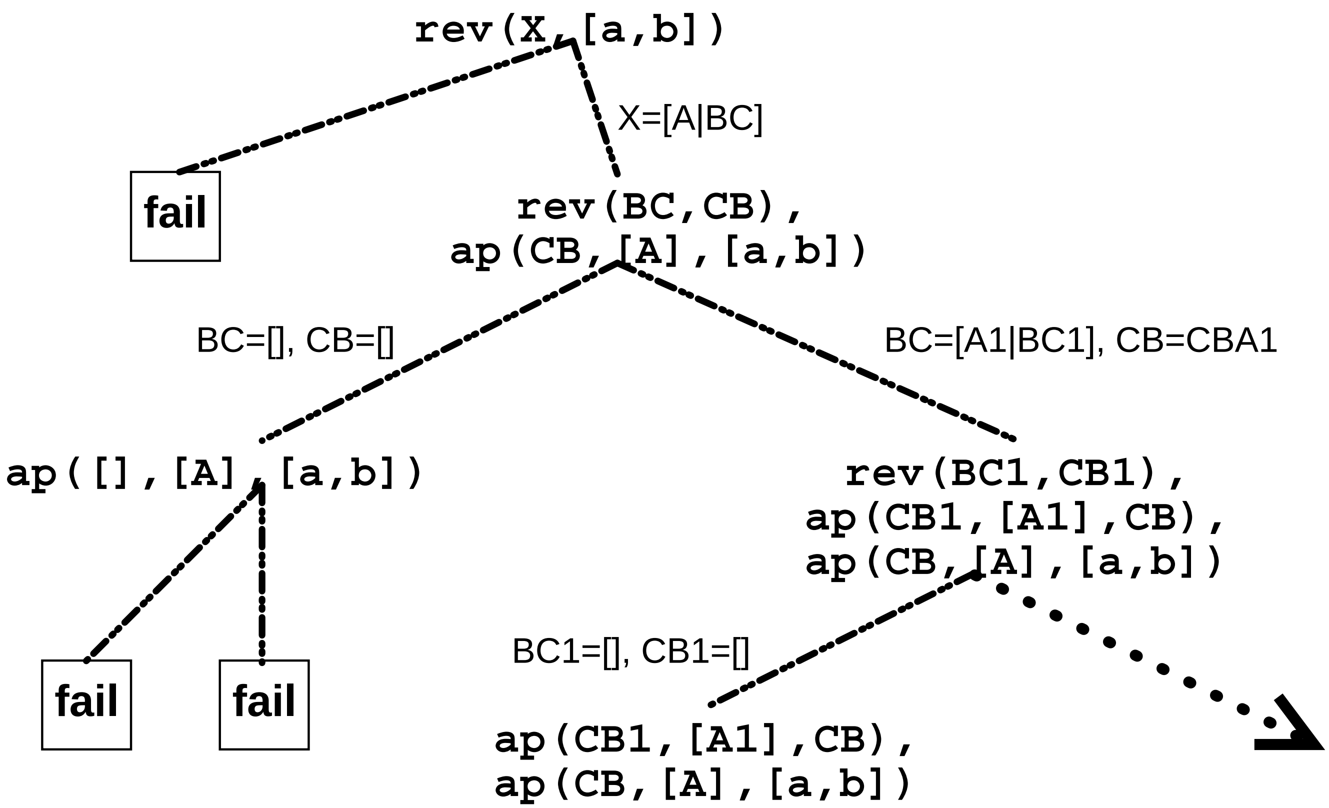
In the "backwards" mode, `rev/2` produces the correct answer, but when we look for more answers, get gets into an infinite loop

Hit control-C then `a` to abort a runaway computation

In the "backwards" mode, the recursive call to `rev/2` *generates*, and call to `append/3` *tests*

The problem: `rev/2` generates *infinitely many* solutions, and `append/3` only accepts one

3.3.3 Reverse Example

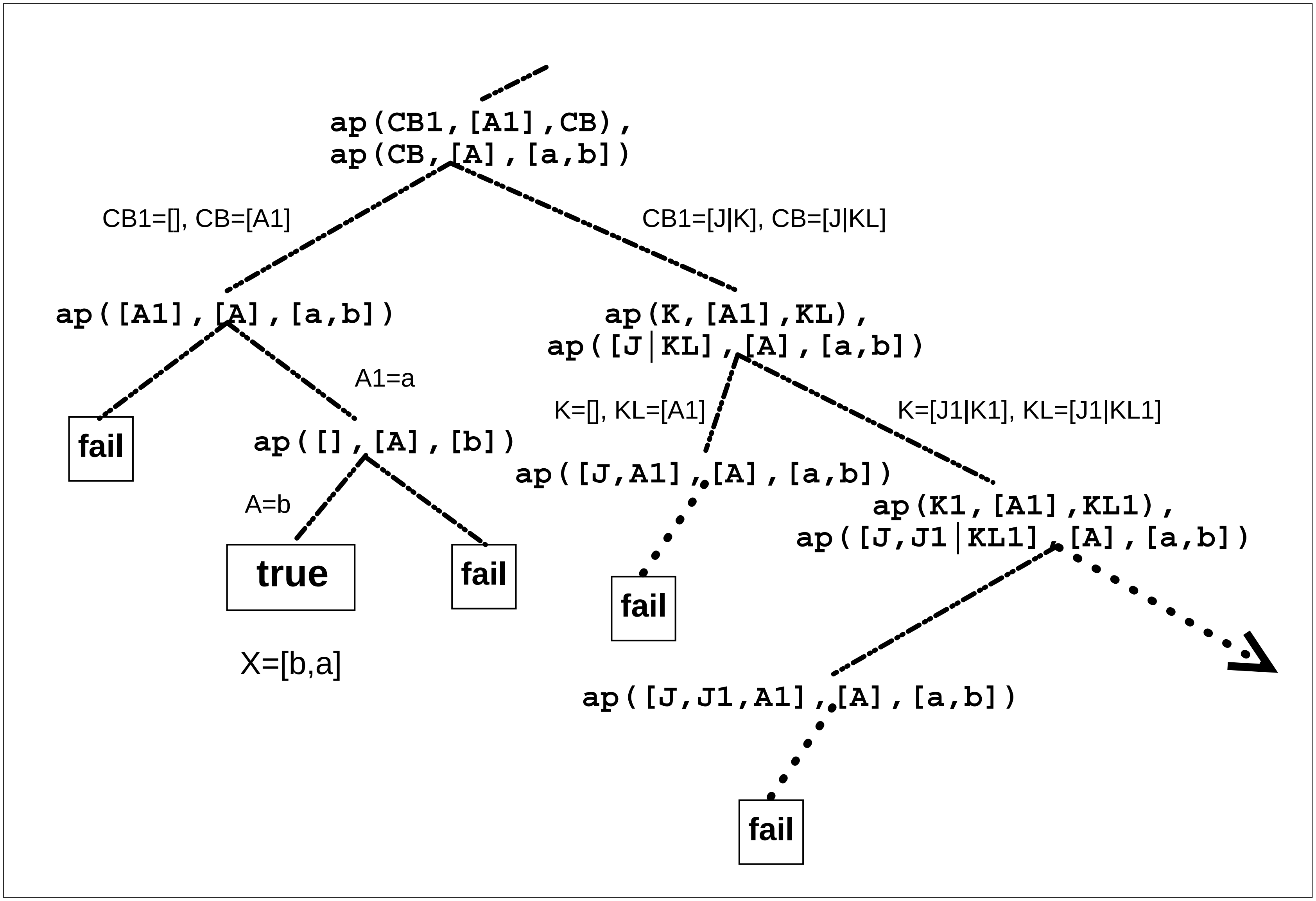


(continued on next slide)

NB: if this succeeds,

X=[A,A1]

Reverse Example (2)



3.3.4 Reverse Body Reordered

Simple solution: reorder the body of the recursive clause

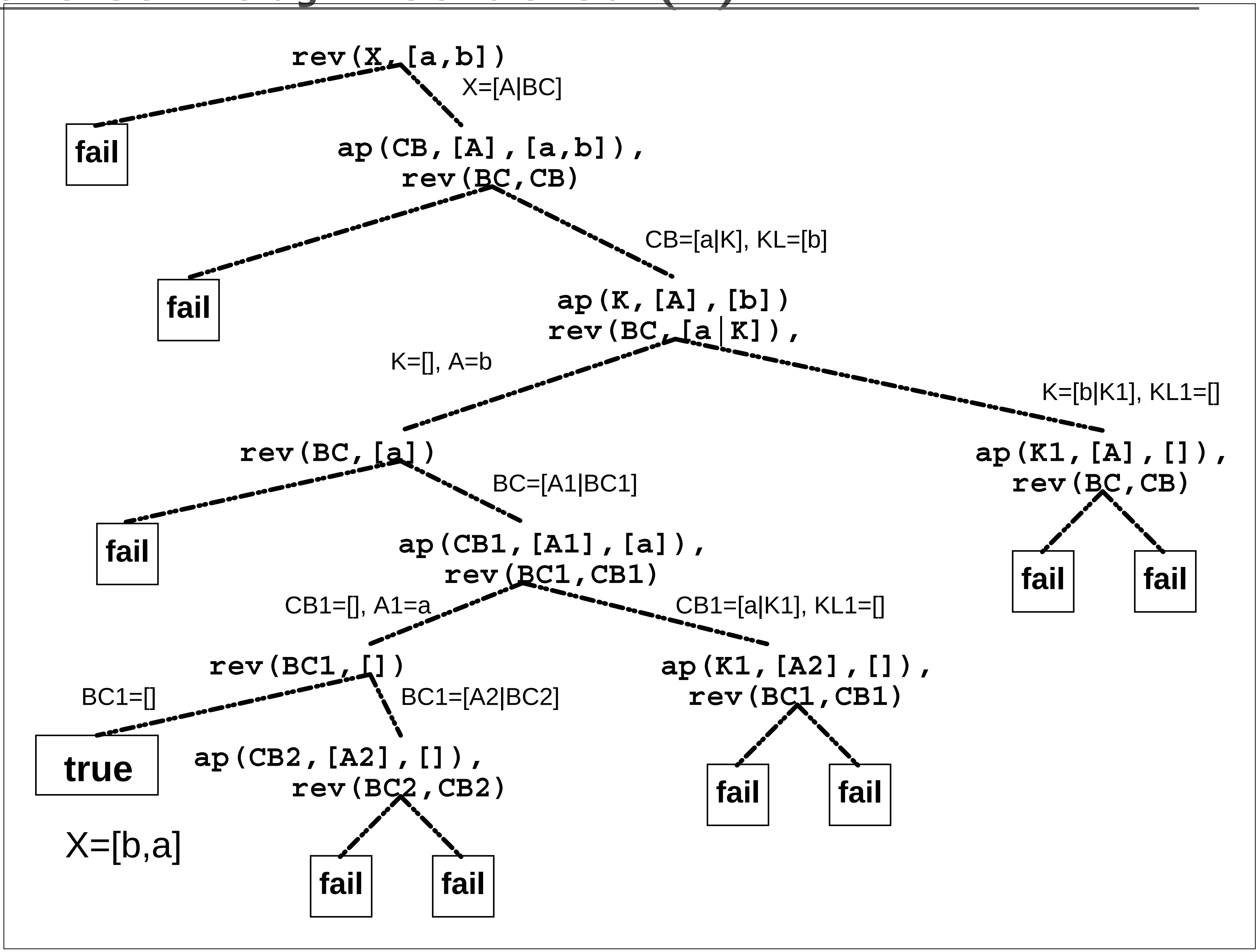
```
rev([], []).             % the reverse of [] is []
rev([A|BC], R) :-        % the reverse of a list [A] ++ BC
  append(CB, [A], R),    %     is CB with A added on the end
  rev(BC, CB).           %     where CB is the reverse of BC
```

This fixes that problem, but causes another:

```
?- rev(R, [a,b,c]).
R = [c, b, a] ;
No
```

```
?- rev([a,b,c], R).
R = [c, b, a] ;
Action (h for help) ? abort
% Execution Aborted
```

Reverse Body Reordered (2)



3.3.5 Fixing Reverse

Solution: make sure both arguments are bound enough to prevent infinite generate and test

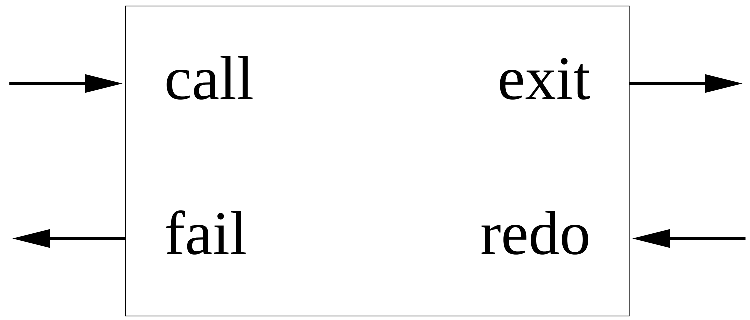
```
reverse(ABC, CBA) :-
        same_length(ABC, CBA), % ensure backbones bound
        rev(ABC, CBA).         % use either old defn

same_length([], []).           % empty lists are same length
same_length([_|As], [_|Bs]) :- % same length if
        same_length(As, Bs).   % tails are same length
```

3.4 The Debugger

Understanding choicepoints and backtracking is essential to understanding Prolog code; debugger is a good way

The Byrd box model can be visualized:



Call port initial entry to the goal
Exit port successful completion of the goal
Redo port backtracking into the goal
Fail port final failure of the goal

Debugger turned on by `trace`, and off with `nodebug`.

3.4.1 Member Example

```
?- trace, member(Z, [a,b]), member(Z, [b,c]).
   Call: (7) member(_G402, [a, b]) ? creep
   Exit: (7) member(a, [a, b]) ? creep
   Call: (7) member(a, [b, c]) ? creep
   Call: (8) member(a, [c]) ? creep
   Call: (9) member(a, []) ? creep
   Fail: (9) member(a, []) ? creep
   Fail: (8) member(a, [c]) ? creep
   Fail: (7) member(a, [b, c]) ? creep
   Redo: (7) member(_G402, [a, b]) ? creep
   Call: (8) member(_G402, [b]) ? creep
   Exit: (8) member(b, [b]) ? creep
   Exit: (7) member(b, [a, b]) ? creep
   Call: (7) member(b, [b, c]) ? creep
   Exit: (7) member(b, [b, c]) ? creep
```

3.4.2 Debugger Commands

Many powerful commands. The most useful are:

h display debugger help

c creep to the next port (enter does the same thing)

s skip over execution; go straight to the exit or fail port

r go back to the initial call port of this goal, undoing all bindings done since starting it; this one is *very* useful

a abort this debugging session level prompt

+ set a spypoint (like a breakpoint) on this predicate

- remove spypoint from this predicate

l leap to the next spypoint

b pause this debugging session and enter a **break level**, with new Prolog prompt; end of file (control-d) reenters debugger.

3.4.3 Debugger Predicates

Prolog also has a few built-in predicates for controlling the debugger.

`spy(Predspec)` Place a spypoint on Predspec, which can be a `Name/Arity` pair, or just a predicate name.

`nospy(Predspec)` Remove the spypoint from Predspec.

`trace` Turn on the debugger

`debug` Turn on the debugger and leap to first spypoint

`nodebug` Turn off the debugger