Resolution

Definition 1. Consider a query

$$?\!-\ \phi_1, \ldots, \phi_n.$$

and a clause

$$\theta :\!-\ \psi_1, \ldots, \psi_m.$$

such that none of the variables in the clause appear in the query. Suppose that the ith formula ϕ_i can be unified with the head θ and let σ be the most general unifier of ϕ_i and θ.

We say that the query can be *resolved* against the clause and that the *resolvent* of the two is the query

$$?\!-\ \phi_1\sigma, \ldots, \phi_{i-1}\sigma,\ \psi_1\sigma, \ldots, \psi_m\sigma,\ \phi_{i+1}\sigma, \ldots \phi_n\sigma.$$

Resolution proof

Prolog works backwards from the goal using resolution steps. The proof succeeds if eventually the query becomes empty.

Consider

```
  ?- likes(colin,Y).
```

The first step is to resolve the query against the clause

```
  likes(X,Y):- cat(X), strokes(Y,X), feeds(Y,X).
```

but note that one of the variables in the clause appears in the query and resolution rules this out.

Renaming variables

The strategy used in Prolog is to choose fresh names for the variables in a clause every time the clause is used.

Thus, we actually resolve against, say,

```
  likes(X1,Y1):- cat(X1), strokes(Y1,X1), feeds(Y1,X1).
```

The renaming of variables becomes very important if the same clause is used twice in a proof.

The resolvent is

```
?- cat(colin), strokes(Y,colin), feeds(Y,colin).
```

The first subgoal resolves immediatedly against a unit clause to give

```
?- strokes(Y,colin), feeds(Y,colin).
```

We can resolve against the clause

```
strokes(Y2,X2):- cat(X2), human(Y2), likes(Y2,X2).
```

to obtain

```
?- cat(colin), human(Y), likes(Y,colin), feeds(Y,colin).
```

The subgoals now can be resolved one-by-one against unit clauses leaving the empty query $\square$ and we are done.

SLD-Resolution

To organize an automatic search for a proof, one must choose:

1. (a) at each step, the subgoal to be considered,

   (b) the program clause to resolve it with;

2. the overall strategy to be used.

The execution of Prolog programs is based on the scheme:

1. (a) select the first subgoal in the list,

   (b) resolve with the program clauses in the order they are listed;

2. use a depth-first search strategy (we'll see this shortly).

This scheme is known as *SLD-resolution* which is short for "Selection driven Linear resolution for Definite clauses".

Example 2. Consider the program

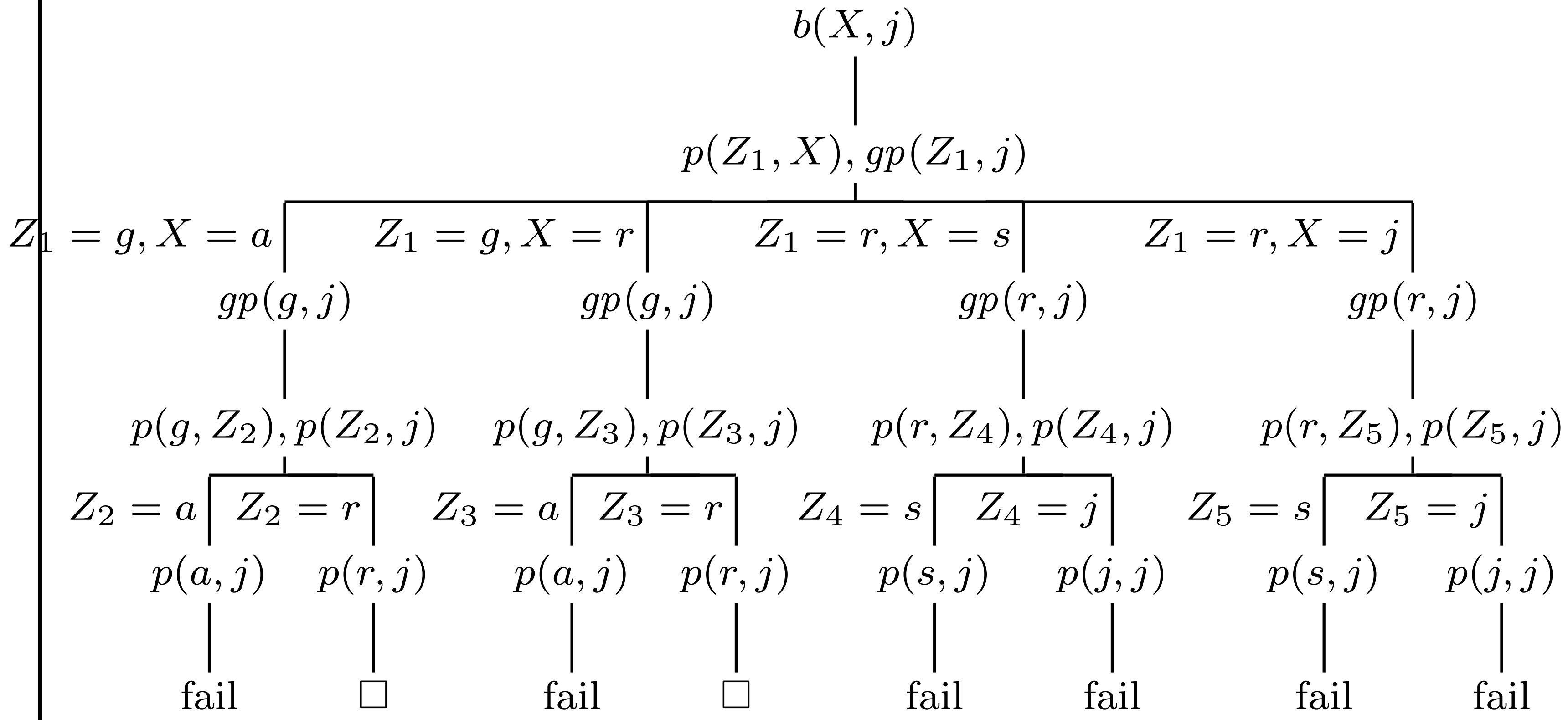
```
parent(g,a).
parent(g,r).
parent(r,s).
parent(r,j).

grandparent(X,Y):- parent(X,Z), parent(Z,Y).

before(X,Y):- parent(Z,X), grandparent(Z,Y).
```

The predicate `before` expresses the relation that one person is in the generation before another.

SLD-trees

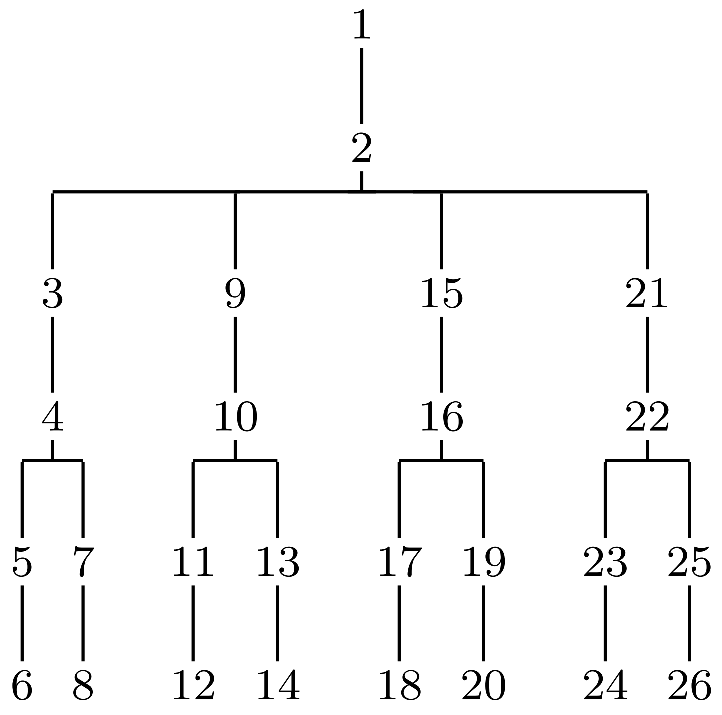
One can picture the search for a proof as a tree of possibilities.

$$\begin{tikzcd}
 & ?\!-\ \mathit{before}(a, Y). \arrow[d, dash] & & & \\
 & ?\!-\ \mathit{parent}(Z_1, a), \mathit{grandparent}(Z_1, Y). \arrow[d, dash, "Z_1 = g"] & & & \\
 & ?\!-\ \mathit{grandparent}(g, Y). \arrow[d, dash] & & & \\
 & ?\!-\ \mathit{parent}(g, Z_2), \mathit{parent}(Z_2, Y). \arrow[dl, dash, "Z_2 = a"'] \arrow[drr, dash, "Z_2 = r"] & & & \\
?\!-\ \mathit{parent}(a, Y). \arrow[d, dash] & & & ?\!-\ \mathit{parent}(r, Y). \arrow[dl, dash, "Y = s"'] \arrow[dr, dash, "Y = j"] & \\
\text{fail} & & \square & & \square
\end{tikzcd}$$


$b(X, j)$
$p(Z_1, X), gp(Z_1, j)$
$Z_1 = g, X = a$
$Z_1 = g, X = r$
$Z_1 = r, X = s$
$Z_1 = r, X = j$
$gp(g, j)$
$gp(g, j)$
$gp(r, j)$
$gp(r, j)$
$p(g, Z_2), p(Z_2, j)$
$p(g, Z_3), p(Z_3, j)$
$p(r, Z_4), p(Z_4, j)$
$p(r, Z_5), p(Z_5, j)$
$Z_2 = a$
$Z_2 = r$
$Z_3 = a$
$Z_3 = r$
$Z_4 = s$
$Z_4 = j$
$Z_5 = s$
$Z_5 = j$
$p(a, j)$
$p(r, j)$
$p(a, j)$
$p(r, j)$
$p(s, j)$
$p(j, j)$
$p(s, j)$
$p(j, j)$
fail
$\square$
fail
$\square$
fail
fail
fail
fail

Depth-first search

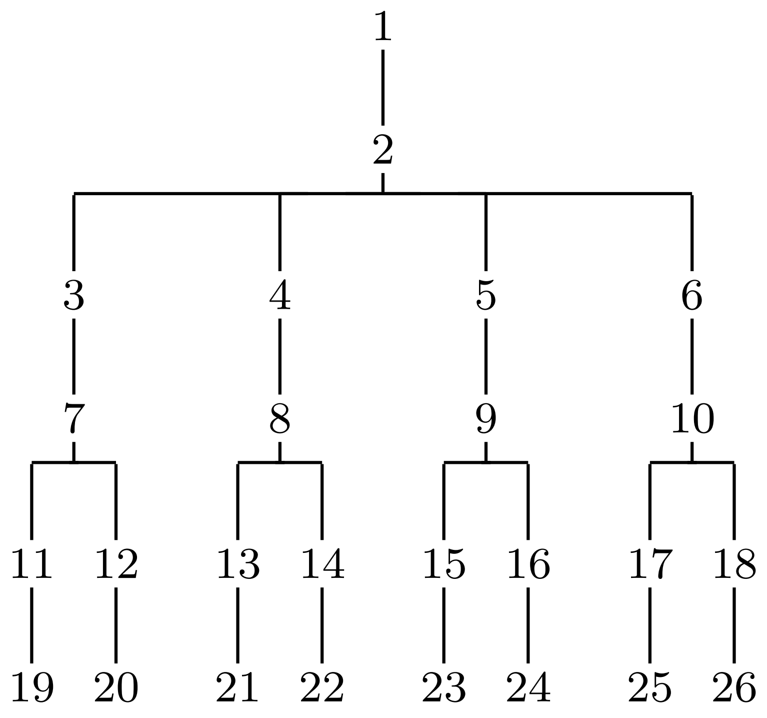
Fully explore each subtree to the left before moving on to the next subtree to the right. The nodes are traversed in the order below.



This is easy to implement without using too much memory.

Breadth-first search

Explore each successive level of the tree before looking at the next level down.



This requires far more memory than a depth-first search but it is nevertheless sometimes appropriate.

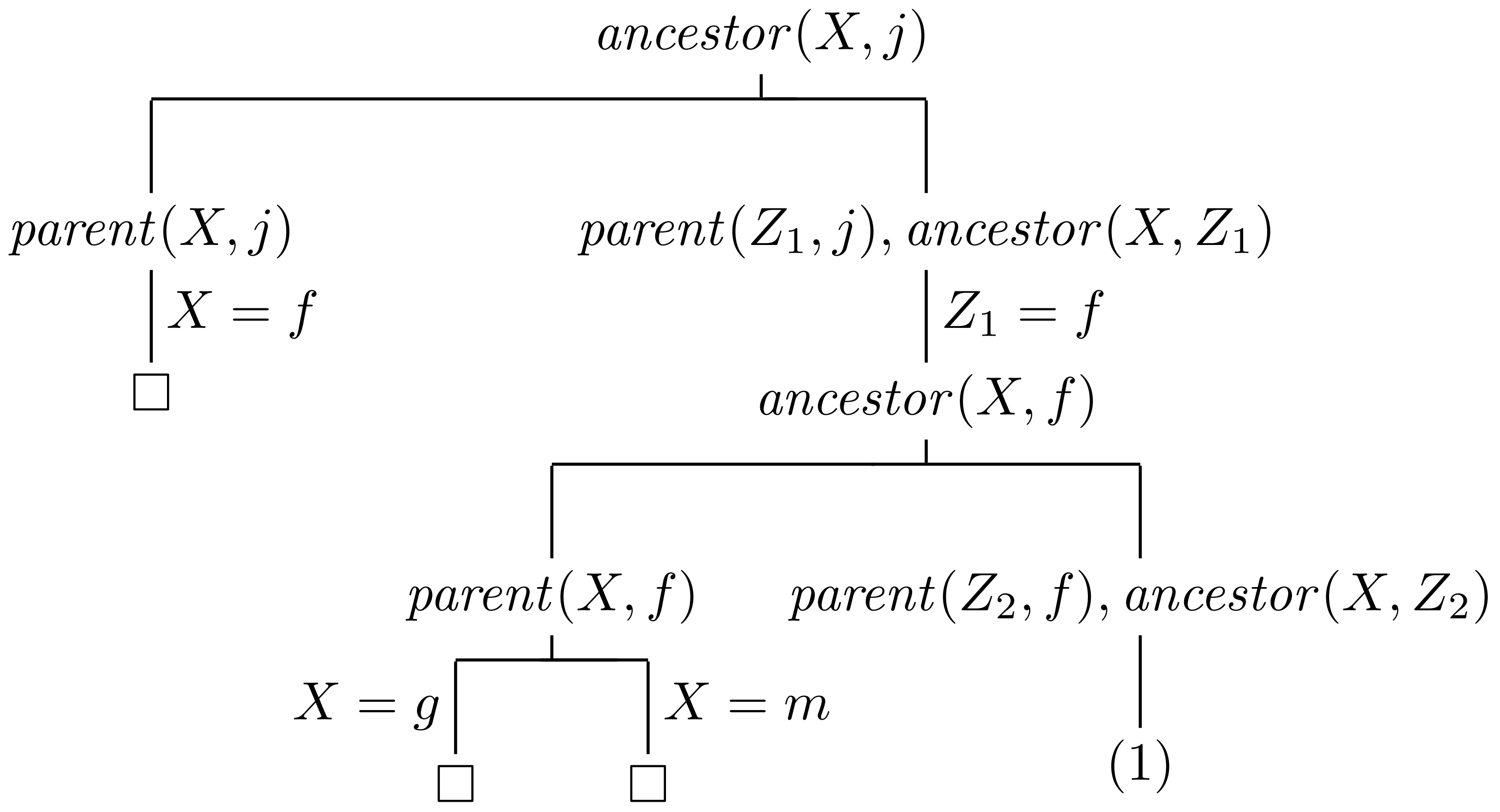
SLD-trees and Recursion

Example 3. Consider the program

```
parent(g,r).
parent(g,a).
parent(r,s).
parent(r,j).

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

with the goal `?- ancestor(X,j).`

The execution can be traced by drawing an SLD-tree,



You might like to check that the branch at (1) leads to failure.

A problem with recursion

The order of the clauses is critical in a recursive program. Suppose that the ancestor program had been written:

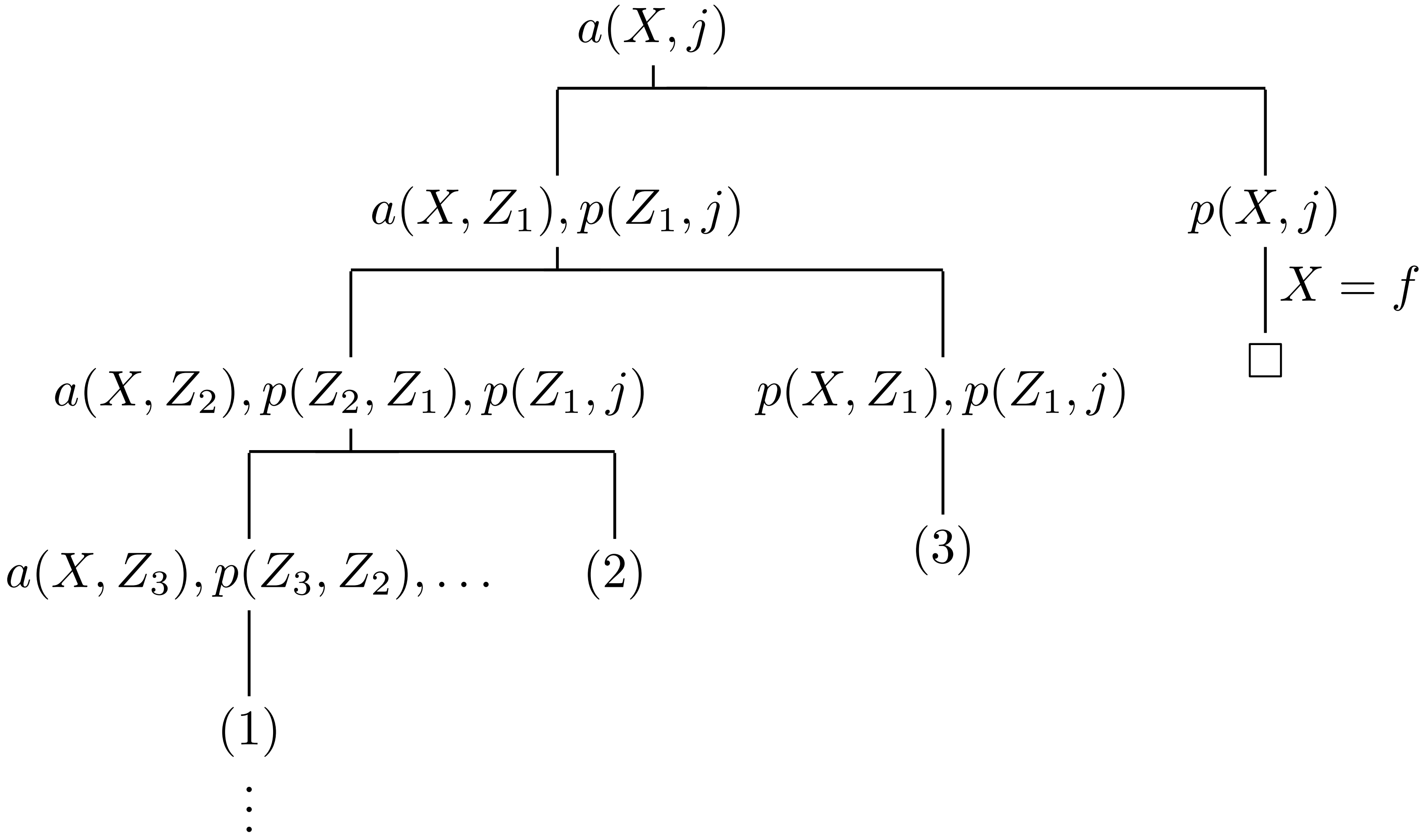
```
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

There is no difference between this and the previous version in purely logical terms — they are equivalent.

However, if we run the query to find the ancestors of `j` then we obtain an error message.

```
?- ancestor(X,j).
[WARNING: Out of local stack]
```

Consider the SLD-tree for the modified program with this goal



Append

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The predicate can be used to join two lists together

```
?- append([b,a],[c,a,d],Zs).
Zs = [b, a, c, a, d]
```

or to split a list into parts

```
?- append(Xs,Ys,[a,b]).
Xs = []
Ys = [a, b] ;
Xs = [a]
Ys = [b] ;
Xs = [a, b]
Ys = []
```

The SLD-tree for the last example.

$$\begin{tikzcd}
& A(Xs,\, Ys, [a,b]) \arrow[dl, dash, "{Xs = [\,],\, Ys = [a,b]}"'] \arrow[dr, dash, "{X = [a|Xs_1]}"] & & \\
\square & & A(Xs_1,\, Ys, [b]) \arrow[dl, dash, "{Xs_1 = [\,],\, Ys = [b]}"'] \arrow[dr, dash, "{Xs_1 = [b|Xs_2]}"] & \\
& \square & & A(Xs_2,\, Ys, [\,]) \arrow[d, dash, "{Xs_2 = [\,],\, Ys = [\,]}"] \\
& & & \square
\end{tikzcd}$$