More on SLD

SLD

- SLD is a SOS
  - States are sequences of atoms
  - Program doesn't change.
- SLD derivation, refutation, computed answer, SLD tree, proof tree

$$A_1,\dots,A_{i-1},A_i,A_{i+1},\dots,A_n \xrightarrow{SLD} (A_1,\dots,A_{i-1},B\rho,A_{i+1},\dots,A_n)\vartheta \quad if \quad \begin{array}{c}(H \leftarrow B) \in P \\ \theta = mgu(A_i, H\rho)\end{array}$$

Example SLD tree

p(X) :- q(X), r(X), s(X).

p(a).

q(a).

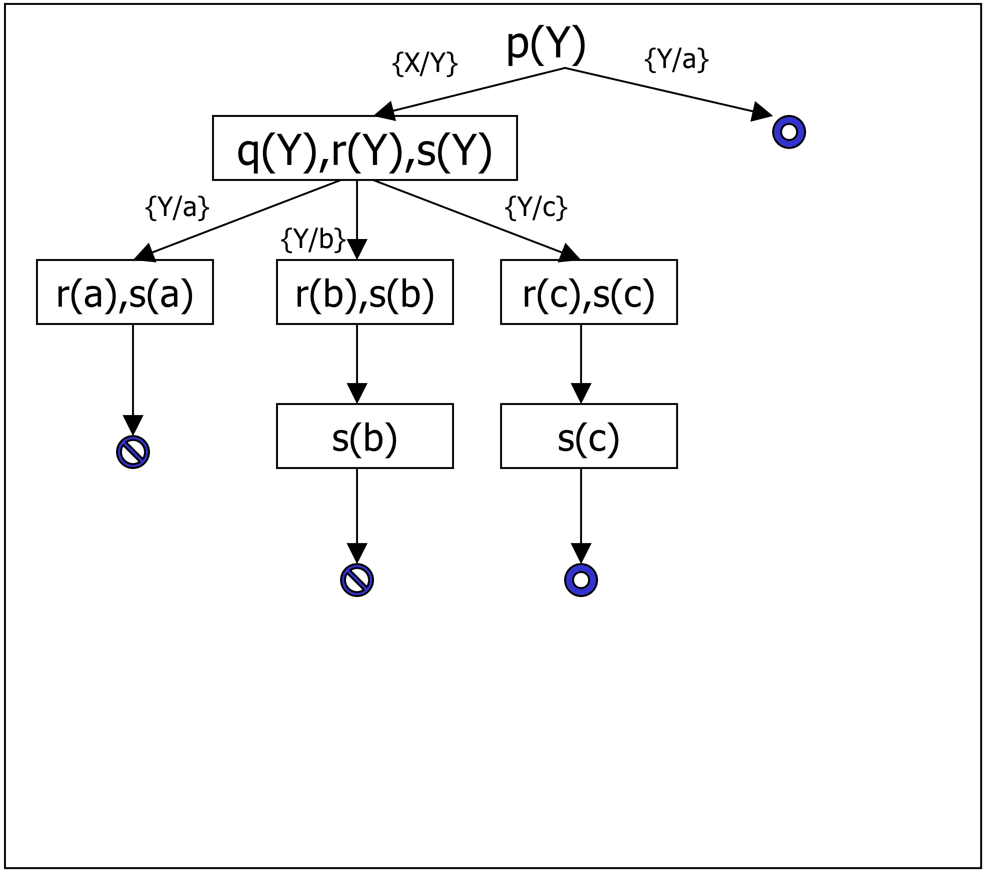
q(b).

q(c).

r(b).

r(c).

s(c).



LD

- Always select the first atom

$$A_1, A_2, \ldots, A_n \xrightarrow{SLD} (B\rho, A_2, \ldots, A_n)\vartheta \quad if \quad \begin{array}{c} (H \leftarrow B) \in P \\ \theta = mgu(A_i, H\rho) \end{array}$$

Alternative SOS

- Substitutions are made explicit.
- Substitutions assign terms to variables occurring in the program rather than their (θ,G) renamed incarnations.
- State is of the form (θ,G) where θ is a substitution and G is a sequence of atoms and special construct which marks the end of the body of a clause.

Alternative SOS

$$(\sigma,(A,G)) \xrightarrow{LD} (\vartheta,(B,mk(A,\sigma,H),G)) \quad if \quad \begin{array}{c} (H \leftarrow B) \in P \\ \theta = mgu(A\sigma\rho, H) \end{array}$$

$$(\vartheta,(mk(A,\sigma,H),G)) \xrightarrow{LD} (\sigma \quad mgu(A\sigma, H\vartheta\rho),G)$$

Prolog programming

Logic operations

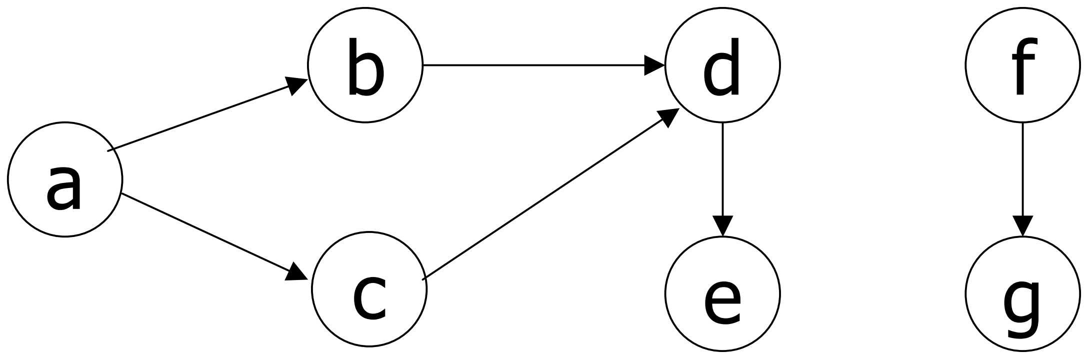
```
inv(0,1).
inv(1,0).

and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).

or(0,0,0).
or(0,1,1).
or(1,0,1).
or(1,1,1).

?- or(X,Y,V1),inv(Z,V2), and(V1,V2,V).
```

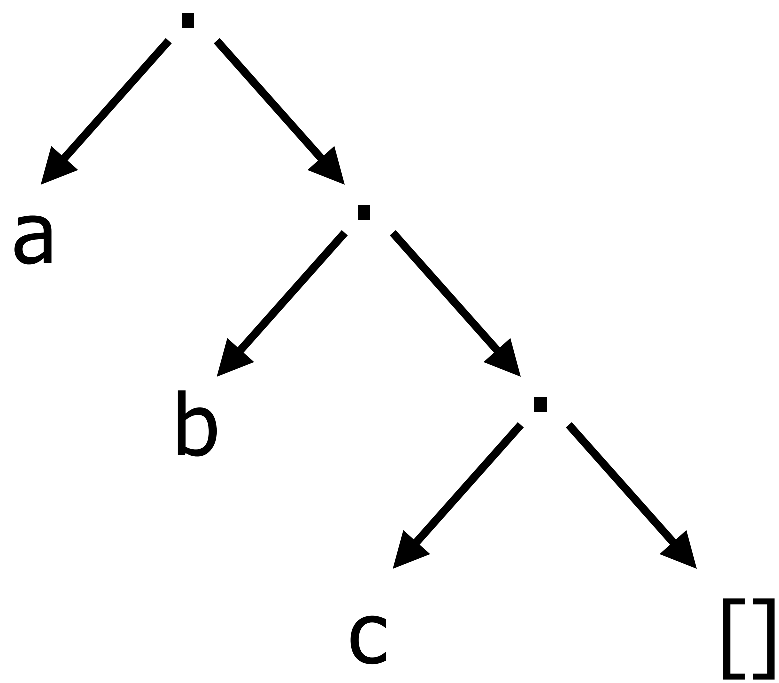
Path




```
edge(a,b).
edge(a,c).
edge(b,d).
edge(c,d).
edge(d,e).
edge(f,g).
```

```
path(N1,N2) :-
        edge(N1,N2).
path(N1,N3) :-
        edge(N1,N2),
        path(N2,N3).
```

List Notation



.(a, .(b, .(c, [])))

More On List Notation

- The empty list: []
- A non-empty list: .(X,Y) or [X|Y]

Syntactic Sugar:

- [b] instead of [b|[]] and .(b, [])
- [a,b] instead of [a|[b]] and [a|[b|[]]]
- [a,b|X] instead of [a|[b|X]]

List manipulation

```
list([ ]).
list([X|Xs]) :- list(Xs).

member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).

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

List Manipulation

```
% reverse(A, B)
% B is A in reverse order
reverse([ ],[ ]).
reverse([X|Xs],Zs) :- reverse(Xs,Ys), append(Ys,[X],Zs).

% Alternative version
reverse(Xs,Ys) :- reverse(Xs,[ ],Ys).

reverse([ ],Ys,Ys).
reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
```

Insertion Sort

```
% sort(A,B)
% B is a sorted version of A
sort([X|Xs],Ys) :- sort(Xs,Zs), insert(X,Zs,Ys).
sort([ ],[ ]).

% insert(A,B,C)
% if  B is a sorted list, then C is sorted
% and contains all elements in B plus A
insert(X,[ ],[X]).
insert(X,[Y|Ys],[Y|Zs]) :- X > Y, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
```

Circuits I

```
circuit(1,1).
circuit(0,0).

circuit(and(X,Y),V) :-
    circuit(X,V1),
    circuit(Y,V2),
    and(V1,V2,V).

circuit(or(X,Y),V) :-
    circuit(X,V1),
    circuit(Y,V2),
    or(V1,V2,V).

circuit(inv(X),V) :-
    circuit(X,V1),
    inv(V1,V).
```

Circuits II

```
circuit(1,1).

circuit(0,0).

circuit(C,V) :-
    nonvar(C),
    C = and(X,Y),
    circuit(X,V1),
    circuit(Y,V2),
    and(V1,V2,V).

circuit(C,V) :-
    nonvar(C),
    C = or(X,Y),
    circuit(X,V1),
    circuit(Y,V2),
    or(V1,V2,V).

circuit(C,V) :-
    nonvar(C),
    C = inv(X),
    circuit(X,V1),
    inv(V1,V).
```

Circuits III

```
circuit(X,X) :-
    var(X),
    !.

circuit(1,1).

circuit(0,0).

circuit(and(X,Y),V) :-
    circuit(X,V1),
    circuit(Y,V2),
    and(V1,V2,V).
```

```
circuit(or(X,Y),V) :-
    circuit(X,V1),
    circuit(Y,V2),
    or(V1,V2,V).

circuit(inv(X),V) :-
    circuit(X,V1),
    inv(V1,V).
```

Cut

p(X) :- q(X), r(X), !, s(X

p(a).

q(a).

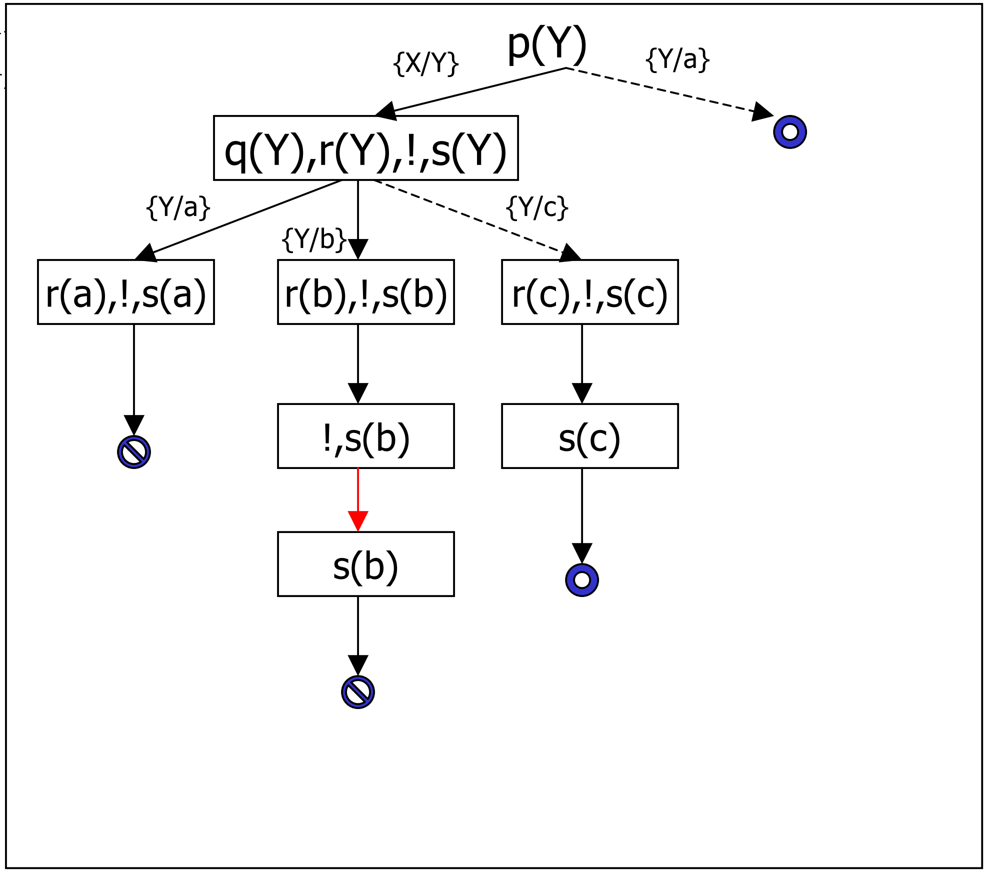
q(b).

q(c).

r(b).

r(c).

s(c).



Meta-Programming

- Meta-programs: programs that takes other programs as inputs such as interpreters, compilers, partial evaluators and debuggers.
- Meta-circular programs: a program in language L that takes programs in L as inputs.
- Meta-interpreter: a program that executes input programs.

A meta-interpreter

- Representation:

A clause

H :- $A_1,\ldots,A_n$

is represented as a fact

pg_clause($H,[A_1,\ldots,A_n]$).

```
solve([]).
solve([A|As]) :-
    solve(A),
    solve(As).
solve(A) :-
    pg_clause(A,B),
    solve(B).
```

A meta-interpreter

- Representation:

  A clause

  $H \text{ :- } A_1,\ldots,A_n$

  is represented as is.
- Prolog implementations provides a built-in predicate *clause*(A,B)

```
solve(true).
solve((A,As)) :-
    solve(A),
    solve(As).
solve(A) :-
    A \= true,
    A \= (_,_),
    clause(A,B),
    solve(B).
```

Ground and non-ground representation

- Previous meta-interpreters use non-ground representation in which a variable is represented by a variable.
- Another variation will be ***ground*** representation. The meta-interpreter then needs to do *renaming*, *unification* and *instantiation* explicitly.

Variations

- Substitutions can be explicitly represented.
- Program can be passed as an argument.
- Proof trees can be constructed and returned.
- Other execution strategies such as breadth-first search can be programmed.
- Other semantics such as OLDT can be programmed.
- Program analyses can be developed as an meta-program/interpreter very fast.

Representation of WHILE programs

x := a	➔ assign(**x**,**a**)
skip	➔ skip
if b then S_1 else S_2	➔ if(**b**, **S_1**,**S_2**)
S_1;S_2	➔ sequent(**S_1**,**S_2**)
while b do S	➔ while(**b**,**S**)
x	➔ var(x)
n	➔ int(***N***(n))
e_1 op e_2	➔ **op**(**e_1**,**e_2**)
op e	➔ **op**(**e**)

Interpreter

```
ae(int(N),State,N).
ae(var(x),State,N) :-
    member((x,N),State).
ae(plus(E,F),State,N) :-
    ae(E,State,Left),
    ae(F,State,Right),
    N is Left + Right.
ae(minus(E),State,N) :-
    ae(E,State,N1),
    N is – N1.
….
```

```
be(true, tt).
be(false,ff).
be(lt(E,F),State,Truth) :-
  ae(E,State,Lft),
  ae(F,State,Rht),
  lt_test(Lft,Rht,Truth).
….

  lt_test(L,R,tt) :- L < R.
  lt_test(L,R,ff) :- R =< L.
```

Interpreter

```
interp(assign(var(X),Exp),Si,So)  :-
    ae(Exp,Si,N), update(Si,X,N,So).
interp(skip,S,S).
interp(sequent(Stm1,Stm2),Si,So) :-
    interp(Stm1,Si,Sm), interp(Stm2,Sm,So).
interp(if(BExp,Stm1,Stm2),Si,So) :-
    be(BExp,Si,Truth),
    cond(Truth,Stm1,Stm2,Si,So).
interp(while(BExp,Stm),Si,So) :-
    be(Bexp,Si,Truth),
   cond(Truth,skip,sequent(Stm,while(BExp,Stm)),Si,So).
```