Declarative Programming

3: logic programming and Prolog
Sentences in definite clause logic: *procedural and declarative meaning*

\[ a \leftarrow b, c. \]

declarative meaning realized by model semantics

to determine whether \( a \) is a logical consequence of the clause, order of atoms in body is irrelevant

procedural meaning realized by proof theory

order of atoms may determine whether \( a \) can be derived

\[ a \leftarrow b, c. \] to prove \( a \), prove \( b \) and then prove \( c \)

\[ a \leftarrow c, b. \] to prove \( a \), prove \( c \) and then prove \( b \)

Imagine \( c \) is false and proof for \( b \) is infinite
Sentences in definite clause logic: *procedural meaning enables programming*

- SLD-resolution refutation
  - procedural knowledge: *how* the inference rules are applied to solve the problem
- algorithm = logic + control
  - declarative knowledge: the *what* of the problem
  - definite clause logic
SLD-resolution refutation: turns resolution refutation into a proof procedure

definite clauses

SLD

linear resolution

selection rule

the clause obtained from a resolution step (the resolvent) is always resolved with a program clause in the next (and not with another resolvent)

defines how to determine how to select a literal to resolve upon and which clause is used when multiple are applicable

left-most

top-down

also: an unwieldy theorem prover in effective programming language

refers to the shape of the resulting proof trees
SLD-resolution refutation:
refutation proof trees based on SLD-resolution

\[
\text{grandfather}(X,Z) :- \text{father}(X,Y), \text{parent}(Y,Z).
\]

\[
\text{parent}(X,Y) :- \text{father}(X,Y).
\]

\[
\text{parent}(X,Y) :- \text{mother}(X,Y).
\]

\[
\text{father}(a,b).
\]

\[
\text{mother}(b,c).
\]

---

\[
\text{grandfather}(a,X) \rightarrow \text{goal (query)}
\]

\[
\text{grandfather}(C,D) :- \text{father}(C,E), \text{parent}(E,D).
\]

\[
\{C/a, D/X\}
\]

\[
\text{father}(a,E), \text{parent}(E,X) \rightarrow \text{derived goal}
\]

\[
\text{father}(a,b).
\]

\[
\{E/b\}
\]

\[
\text{parent}(b,X) \rightarrow \text{computed substitution}
\]

\[
\text{parent}(U,V) :- \text{mother}(U,V).
\]

\[
\{U/b, V/X\}
\]

\[
\text{mother}(b,X) \rightarrow \text{computed answer substitution}
\]

\[
\text{mother}(b,c).
\]

\[
\{X/c\}
\]

\[
\{X/c, C/a, D/c, E/b, U/b, V/c\}
\]
SLD-resolution refutation: 
SLD-trees

grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).

Every leaf corresponds to a successful refutation (a success branch). A blocked leaf corresponds to a failed branch.

Prolog does a depth-first traversal of an SLD tree.

What if an SLD tree has infinite branches?

Prolog traverses SLD-trees depth-first, backtracking from a blocked node to the last choice point (also from a success node when more answers are requested).

every path from the query root to the empty clause corresponds to a proof tree (a successful refutation proof)
Problems with SLD-resolution refutation: *never reaching success branch because of infinite subtrees*

```prolog
sibling(X,Y) :- sibling(Y,X).
sibling(b,a).

:- sibling(a,X)  
  :- sibling(X,a)  
  :- sibling(a,X)  
  :- sibling(X,a)  
...  
```

had we re-ordered the clauses, we would have reached a success branch at the second choice point.

incompleteness of Prolog is a design choice: **breadth-first traversal** would require keeping all resolvents on a level in memory instead of 1.

Prolog loops on this query; renders it incomplete! only because of **depth-first traversal** and not because of resolution as all answers are represented by success branches in the SLD-tree.
Problems with SLD-resolution refutation: Prolog loops on infinite SLD-trees when no (more) answers can be found

\[
\text{sibling}(a,b).
\text{sibling}(b,c).
\text{sibling}(X,Y) :- \text{sibling}(X,Z), \text{sibling}(Z,Y).
\]

\[
\text{:-sibling}(a,X)
\]

\[
\quad \text{:-sibling}(a,Z), \text{sibling}(Z,Y)
\]

\[
\quad \text{:-sibling}(a,U), \text{sibling}(U,Z), \text{sibling}(Z,Y)
\]

\[
\quad \text{:-sibling}(b,Y)
\]

\[
\quad \text{:-sibling}(a,Z), \text{sibling}(Z,Y)
\]

\[
\cdots
\]

Cannot be helped using breadth-first traversal: is due to \textbf{semi-decidability} of full and definite clausal logic.
Problems with SLD-resolution refutation: illustrated on list generation

Prolog would loop without finding answers if clauses were reversed!

```
list([]).
list([H|T]):-list(T).

?-list(L).
L = [];
L = [A];
L = [A,B];
...
```

benign: infinitely many lists of arbitrary length are generated
Problems with SLD-resolution refutation: illustrated on list generation

\[\text{plist}([]).\]
\[\text{plist([H|T]):=p(H),plist(T).}\]
\[\text{p(1).}\]
\[\text{p(2).}\]

\[\text{?-plist(L).}\]
\[\text{L=[];}\]
\[\text{L=[1];}\]
\[\text{L=[1,1];}\]
\[\text{...}\]

less benign: only lists containing 1s are generated

explored by Prolog

success branches that are never reached
SLD-resolution refutation: implementing backtracking

when a failure branch is reached (non-empty resolvent which cannot be reduced further), next alternative for the last-chosen program clause has to be tried.

requires remembering previous resolvents for which not all alternatives have been explored together with the last program clause that has been explored at that point.

backtracking =
  popping resolvent from stack and exploring next alternative.
Pruning the search by means of cut: cutting choice points

need to be remembered for all resolvents for which not all alternatives have been explored
unnecessary alternatives will eventually be explored

parent(X,Y):-father(X,Y).
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).

?-parent(john,C)

:-father(john,C) :-mother(john,C)

[ ]
at this point, we know that exploring the alternative clause for parent/2 will fail

parent(X,Y):-father(X,Y),!.
parent(X,Y):-mother(X,Y).
father(john,paul).
mother(mary,paul).

?-parent(john,C)

:-father(john,C),!

:-mother(john,C)

[ ]
tells Prolog that this is the only success branch
Pruning the search by means of cut: operational semantics

“Once you’ve reached me, stick with all variable substitutions you’ve found after you entered my clause”

Prolog won’t try alternatives for:
- literals left to the cut
- nor the clause in which the cut is found

A cut evaluates to true.
Pruning the search by means of cut: an example

\[ \text{p(X,Y)} :- \text{q(X,Y)}. \]
\[ \text{p(X,Y)} :- \text{r(X,Y)}. \]
\[ \text{q(X,Y)} :- \text{s(X)}, !, \text{t(Y)}. \]
\[ \text{r(c,d)}. \]
\[ \text{s(a)}. \]
\[ \text{s(b)}. \]
\[ \text{t(a)}. \]
\[ \text{t(b)}. \]

Are not yet on the stack when cut is reached.

no pruning for literals right to the cut

no pruning above the head of the clause containing the cut

Are not yet on the stack when cut is reached.

\( \text{p(X,Y)} \)
\( \text{q(X,Y)} \)
\( \text{r(X,Y)} \)
\( \text{s(X)} \)
\( \text{t(Y)} \)

[ ]

[ ]

[ ]

[ ]

[ ]

[ ]
Pruning the search by means of cut: different kinds of cut

**green cut**

- does not prune away success branches

- stresses that the conjuncts to its left are deterministic and therefore do not have alternative solutions

- and that the clauses below with the same head won’t result in alternative solutions either

**red cut**

- prunes success branches

- some logical consequences of the program are not returned

- has the declarative and procedural meaning of the program diverge
Pruning the search by means of cut: red cuts

\[
\text{parent}(X,Y) :\neg\text{father}(X,Y), !. \\
\text{parent}(X,Y) :\neg\text{mother}(X,Y). \\
\text{father}(\text{john}, \text{paul}). \\
\text{father}(\text{john}, \text{peter}). \\
\text{mother}(\text{mary}, \text{paul}). \\
\text{mother}(\text{mary}, \text{peter}). \\
\]

\{C/peter\}

\[
\text{same query, but John has multiple children in this program}
\]

\[
\text{parent}(X,Y) :\neg\text{father}(X,Y), !. \\
\text{parent}(X,Y) :\neg\text{mother}(X,Y). \\
\text{father}(\text{john}, \text{paul}). \\
\text{mother}(\text{mary}, \text{paul}). \\
\]

\{P/mary\}

\[
\text{same program, but query quantifies over parents rather than children}
\]

?-\text{parent}(\text{john}, C)

\[
\begin{align*}
\text{:-}\text{father}(\text{john}, C), !. \\
\text{:-}\text{mother}(\text{john}, C) \\
\end{align*}
\]

\[
\text{the cut is now red as a success branch is pruned}
\]

?-\text{parent}(P, \text{paul})

\[
\begin{align*}
\text{:-}\text{father}(P, \text{paul}), !. \\
\text{:-}\text{mother}(P, \text{paul}) \\
\end{align*}
\]

\[
\text{the cut is only green when the literal to its left is deterministic}
\]
Pruning the search by means of cut: placement of cut

likes(peter,Y):-! friendly(Y).
likes(T,S):-student_of(S,T).
student_of(maria,peter).
student_of(paul,peter).
friendly(maria).

?-likes(A,B)
![]
A=peter
B=maria
!
A=peter
B=paul

?:-student_of(B,A)
![]
A=peter
B=maria
![]
A=peter
B=paul

?-likes(A,B)
![]
A=peter
B=maria
![]
A=peter
B=paul

likes(peter,Y):-! friendly(Y).
likes(T,S):-student_of(S,T),!.
Pruning the search by means of cut: 
more dangers of cut

max(M,N,M) :- M>=N.
max(M,N,N) :- M=<N.

clauses are not mutually exclusive
two ways to solve query \texttt{?\-max(3,3,5)}

max(M,N,M) :- M>=N,!.
max(M,N,N).

could use red cut to prune second way

\textbf{problem:}
\texttt{?\-max(5,3,3)}
succeeds

\textbf{Better to use}
\texttt{>= and <}
Negation as failure: specific usage pattern of cut

\[
p :- q, !, r.
p :- s.
\]

Only tried when \(q\) fails

\[
?-p
\]

Such uses are equivalent to the higher-level

\[
p :- q, !, r.
p :- \text{not}_q, s.
\]

Where

\[
\text{not}_q :- q, !, \text{fail}.
\text{not}_q.
\]

Built-in predicate always false

Prolog’s not/1 meta-predicate captures such uses:

\[
\text{not}(\text{Goal}) :- \text{Goal}, ! \text{fail}.
\text{not}(\text{Goal}).
\]

Slight abuse of syntax equivalent to call(\text{Goal})

\[
\text{not}(\text{Goal}) \text{ is proved by failing to prove Goal}
\]

In modern Prologs: use \(+\) instead of not
Negation as failure: SLD-tree where not(q) succeeds because q fails

p :- q, r.
p :- not(q), s.
s.

not(Goal) :- Goal, !, fail.
not(Goal).

?- p

:- q, r

:- not(q), s

:- q, !, fail, s

:- s

q evaluated twice

version with ! was more efficient, but uses of not/1 are easier to understand
Negation as failure:  
*SLD-tree where not(q) fails because q succeeds*

\[
p \leftarrow \text{not}(q), r.  
p \leftarrow q.  
q.  
r.  
\]

\[
\text{not}(\text{Goal}) \leftarrow \text{Goal}, !, \text{fail}.  
\text{not}(\text{Goal}).  
\]

branch corresponding to second clause of not/1 is pruned
Negation as failure: floundering occurs when argument is not ground

\[
bachelor(X) :- \neg not(married(X)), man(X).
\]

\[
\text{man(fred). man(peter). married(fred).}
\]

unintentionally interpreted as “X is a bachelor if nobody is married and X is man”

query has no answers

\[
?-bachelor(X)
\]

\[
:-not(married(X)), man(X)
\]

\[
:-married(X), !, fail, man(X)
\]

\[
:-!, fail, man(fred)
\]

\[
:-\neg fail, man(fred)
\]

these are the bachelors we were looking for!

\[
\text{not(Goal)} :- \neg Goal, !, fail.
\]

\[
\text{not(Goal)}.
\]
Negation as failure: avoiding floundering

Correct implementation of SLDNF-resolution:

\[ \text{not}(\text{Goal}) \text{ fails only if } \text{Goal} \text{ has a refutation with an empty answer substitution} \]

Prolog does not perform this check:

not(married(X)) failed because married(X) succeeded with \{X/fred\}

Work-around: If Goal is ground, only empty answer substitutions are possible

bachelor(X):- man(X),
            not(married(X)).
man(fred).
man(peter).
married(fred).
Negation as failure: avoiding floundering

?- bachelor(X)

:- man(X), not(married(X))

:- not(married(fred))

:- married(fred), !, fail

:- !, fail

:- fail

:- not(married(peter))

:- married(peter), !, fail

bachelor(X):- man(X), not(married(X)).

man(fred).
man(peter).
married(fred).