CMSC 330: Organization of Programming Languages

Prolog Advanced Topic: Cut
Prolog Terminology

- A query, goal, or term where variables do not occur is called **ground**; else it’s **nonground**
  - foo(a,b) is ground; bar(X) is nonground

- A substitution $\theta$ is a partial map from variables to terms where $\text{domain}(\theta) \cap \text{range}(\theta) = \emptyset$
  - Variables are terms, so a substitution can map variables to other variables, but circularly

- A is an **instance** of B if there is a substitution such that $A = B\theta$  

- C is a **common instance** of A and B if it is an instance of A and an instance of B

The substitution $\theta$ applied to B
Quick Quiz

- Which of these are ground terms?
  - jedi(luke)  
    - ground
  - jedi(yoda)  
    - ground
  - sith(X)  
    - not ground

- Which of these is an instance of fight(A,B)?
  - jedi(luke)  
    - no (heads don’t match)
  - fight(C,D)  
    - yes, \( \theta = \{ A \rightarrow C, B \rightarrow D \} \)
  - fight(A,luke)  
    - no (A \rightarrow A not allowed)
  - fight(luke,yoda)  
    - yes, \( \theta = \{ A \rightarrow \text{luke}, B \rightarrow \text{yoda} \} \)
Prolog’s Algorithm \text{Solve}() \\

\text{Solve}(\text{goal } G, \text{ program } P, \text{ substitution } \theta) = \\
\begin{itemize}
  \item Suppose \( G \) is \( A_1, \ldots, A_n \). Choose goal \( A_1 \).
  \item For each clause \( A : - B_1, B_2, \ldots, B_k \) in \( P \), 
    \begin{itemize}
      \item if \( \theta_1 \) is the mgu of \( A \) and \( A_1 \theta \) then
        \begin{itemize}
          \item If \( \text{Solve}(\{B_1, \ldots, B_k, A_2, \ldots, A_n\}, P, \theta \cdot \theta_1) = \text{some } \theta' \) then return \( \theta' \)
          \item (else it has failed, so we continue the for loop)
        \end{itemize}
      \item (else unification has failed, so try another rule)
    \end{itemize}
  \item If loop exits return \text{fail}
  \item Output: \( \theta \) s.t. \( G\theta \) can be deduced from \( P \), or fail
\end{itemize}

\begin{itemize}
  \item Starts as empty
  \item Chooses goals in order
  \item Most General Unifier
  \item Implements backtracking
\end{itemize}
Example

on_vacation(mary).
on_vacation(peter).
has_money(peter).

travel(X) :- on_vacation(X),
          has_money(X).

?= travel(Y).
  • \( \theta = X \rightarrow Y_1, Y \rightarrow Y_1 \)
  • on_vacation(X), has_money(X)
  • \( \theta = X \rightarrow Y_1, Y \rightarrow Y_1, Y_1 \rightarrow \text{mary} \)
  • has_money(X)
    • \( X = \text{mary} \) fails. Backtrack.
    • on_vacation(X), has_money(X)
      • \( \theta = X \rightarrow Y_1, Y \rightarrow Y_1, Y_1 \rightarrow \text{peter} \)
      • has_money(X)
        • \( X = \text{peter} \) succeeds with \( \theta = X \rightarrow Y_1, Y \rightarrow Y_1, Y_1 \rightarrow \text{peter} \)
        • Output is thus \( Y = \text{peter} \).
Some Additional Built-in Predicates

- “Consulting” (loading) programs
  ```prolog
  ?- consult('file.pl')  
  ?- ['file.pl']  
  ?- [file]
  ```

- Output/Input
  ```prolog
  ?- write('Hello world'), nl  
  ?- read(X).
  ```

- (Dynamic) type checking
  ```prolog
  ?- atom(elephant)  
  ?- atom(Elephant)
  ```

- help
  ```prolog
  ```

- fail and true
When a ! is reached, it succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause the cut occurs in.

- Suppose we have clause C which is $A : B_1, \ldots, B_k, !, \ldots, B_n$.
- If the current goal unifies with $A$, and $B_1, \ldots, B_k$ further succeed, the program is committed to the choice of C for the goal.
  - If any $B_i$ for $i > k$ fail, backtracking only goes as far as the cut.
  - If the cut is reached when backtracking, the goal fails.
Cut

- Limits backtracking to predicates to right of cut

- Example

  jedi(luke).
  jedi(yoda).
  sith(vader).
  sith(maul).
  fight2(X,Y) :- jedi(X), !, sith(Y).
  fight3(X,Y) :- jedi(X), sith(Y), !.

  ?- fight2(A,B).
  A=luke,
  B=vader;
  A=luke,
  B=maul.

  ?- fight3(A,B).
  A=luke,
  B=vader.
Prolog Search Tree Limited By Cut

?- fight2(A,B).

A=X, B=Y

?- jedi(X),!,sith(Y).

X=luke

?- jedi(luke),!,sith(Y).

Y=vader

?- sith(vader).

Y=maul

?- sith(maul).

X=yoda

?- jedi(yoda),sith(Y).

Y=vader

?- sith(vader).

Y=maul

?- sith(maul).
What Exactly Is Cut Doing?

- Prunes all clauses below it
- Prunes alternative solutions to its left
- Does not affect the goals to its right

Note: Cut only affects this call to merge. Does not affect backtracking of functions calling merge, or later recursive call to merge past cut.
Quiz 1: What does this query return?

\[
s(c).
s(m).
s(d).
solve(X) :- s(X), !.
solve(other_solution).
?– solve(X).
\]

A. \(X = c; X=m; X = d; X = \text{other\_solution}\).
B. \(X = c\)
C. \(X = c; X=m; X = d;\)
D. \text{true}\
Quiz 1: What does this query return?

\[
\text{s(c).}
\text{s(m).}
\text{s(d).}
\text{solve(X) :- s(X), !.}
\text{solve(other\_solution).}
?\text{- solve(X).}
\]

A. \( X = \text{c}; X = \text{m}; X = \text{d}; X = \text{other\_solution}. \)
B. \( X = \text{c} \)
C. \( X = \text{c}; X = \text{m}; X = \text{d}; \)
D. \( \text{true} \)
Quiz 2: What does this query return?

```
check(_, []) :- !.
check(E, [H|T]) :- E > H, check(E, T).
?- check(10, [4, 3, 2]).
```

A. false.
B. true; false.
C. true.
D. false; true.
Quiz 2: What does this query return?

check(_, []) :- !.
check(E, [H|T]) :- E > H, check(E, T).
?- check(10, [4, 3, 2]).

A. false.
B. true; false.
C. true.
D. false; true.
Why Use Cuts?

- Save time and space, or eliminate redundancy
  - Prune useless branches in the search tree
    - If sure these branches will not lead to solutions
  - These are green cuts

- Guide the search to a different solution
  - Change the meaning of the program
    - Intentionally returning only subset of possible solutions
  - These are red cuts
Quiz 3: Is this a green or red cut?

\[\text{s(c).} \]
\[\text{s(m).} \]
\[\text{s(d).} \]
\[\text{solve(X) :- s(X), !.} \]
\[\text{solve(other_solution).} \]
\[?\text{- solve(X).} \]

A. Green
B. Red
Quiz 3: Is this a green or red cut?

s(c).
s(m).
s(d).

solve(X) :- s(X), !.
solve(other_solution).
?- solve(X).

A. Green
B. Red
Quiz 4: Is this a green or red cut?

check(_, []) :- !.
check(E, [H|T]) :- E > H, !, check(E, T).
?- check(10, [4, 3, 2]).

A. Green
B. Red
Quiz 4: Is this a green or red cut?

check(_, []) :- !.
check(E, [H|T]) :- E > H, !, check(E, T).
?- check(10, [4, 3, 2]).

A. Green
B. Red
Quiz 5: Is this a green or red cut?

if_then_else(P,Q,__) :- P, !, Q.
if_then_else(__,__,R) :- R.

A. Green
B. Red
Quiz 5: Is this a green or red cut?

if_then_else(P,Q,_):- P, !, Q.
if_then_else(_,_,R):- R.

A. Green
B. Red
Negation As Failure

- (Red) cut used to implement negation (not)
- Example
  
  \[
  \text{not}(X) :\text{-} \text{call}(X), !, \text{fail}.
  \]
  
  \[
  \text{not}(X).
  \]

- If \( X \) succeeds, then the cut is reached, committing it; \text{fail} causes the whole thing to fail.
- If \( X \) fails, then the second rule is reached, and the overall goal succeeds.
  - FYI, \( X \) here refers to an arbitrary goal
  - Effect of not depends crucially on rule order
Not

- Not is tricky to use
  - Does not mean “not true”
  - Just means “not provable”

- Example
  jedi(luke).
  jedi(vader).
  sith(vader).

  Cannot prove either jedi(leia) or sith(leia) are true, so not( ) returns true

?- not(sith(luke)).
true.

?- not(sith(vader)).
false.

?- not(jedi(leia)).
true.

?- not(sith(leia)).
true.

not(X) :- X, !, fail.
not(X).
Not (cont.)

- Not is tricky to use
  - Does not mean “not true”
  - Just means “not provable”

- Example
  - `jedi(luke).`
  - `jedi(vader).`
  - `sith(vader).`

```
not(X) :- X, !, fail.
not(X).
```

```
?- not(sith(X)).
false.
```

Huh? Why not return X=luke?

Because not(sith(X)) does not mean “Can prove sith(X) is false for some X”

```
not(sith(X)) :- sith(X), !, fail.
not(sith(X)).
```

Instead, it means “Cannot prove sith(X) is true for some X”. So X=vader causes not(sith(X)) to fail and return false
Not – Search Tree

jedi(luke).

jedi(vader).
sith(vader).

not(sith(X)) :- sith(X), !, fail.
not(sith(X)).

Will search for all X such that sith(X) is true.
Not (cont.)

- Ordering of clauses matters
- Example
  
  ```prolog
  jedi(luke).
  jedi(vader).
  sith(vader).
  true_jedi1(X) :-
      jedi(X), not(sith(X)).
  true_jedi2(X) :-
      not(sith(X)), jedi(X).
  ```

  
  ```prolog
  not(X) :- X, !, fail.
  not(X).
  ```

  X=vader causes not(sith(X)) to fail; Will not backtrack to X=luke, since sith(luke) is not a fact
Will search for all X such that sith(X) is true.

not(sith(vader)) fails
Not and \(\neq\)

- **Built-in operators**
  - \(+\) is not
  - \(X \neq Y\) is same as \(\text{not}(X=Y)\)
  - \(X \neq= Y\) is same as \(\text{not}(X==Y)\)

- **So be careful using \(\neq\)**
  - Ordering of clauses matters
  - Try to ensure operands of \(\neq\) are instantiated
Example Using \( \neq \)

- **Example**
  
  ```prolog
  jedi(luke).
  jedi(yoda).
  help2(X,Y) :- jedi(X), jedi(Y), X \( \neq \) Y.
  help3(X,Y) :- jedi(X), X \( \neq \) Y, jedi(Y).
  help4(X,Y) :- X \( \neq \) Y, jedi(X), jedi(Y).
  ```

- **Example Using \( \neq \)**

  ```prolog
  not(X) :- X, !, fail.
  not(X).
  ```

- **Example Using \( \neq \)**

  ```prolog
  help2(X,Y).  
  X=luke,  
  Y=yoda;
  ```

- **Example Using \( \neq \)**

  ```prolog
  help3(X,Y).  
  X=yoda,  
  Y=luke.
  ```

- **Example Using \( \neq \)**

  ```prolog
  help2(X,Y).  
  X=yoda,  
  Y=luke;
  ```

- **Example Using \( \neq \)**

  ```prolog
  help3(X,Y).  
  false.
  ```

- **Example Using \( \neq \)**

  ```prolog
  After selecting X, can choose Y=X and fail X \( \neq \) Y.
  ```
Help3 – Search Tree

\[
\text{not}(X=Y) :- X=Y, !, \text{fail}.
\]

\[
\text{help3}(X,Y) :- \begin{array}{c}
\text{jedi}(X), \quad X \not= Y, \\
\text{jedi}(Y).
\end{array}
\]

\[
\text{not}(X=Y).
\]

\[
\text{jedi}(\text{luke}).
\]

\[
\text{jedi}(\text{yoda}).
\]

\[
?- \text{jedi}(\text{luke}), \quad \text{luke} \not= Y, \quad \text{jedi}(Y).
\]

\[
?- \text{jedi}(\text{yoda}), \quad \text{yoda} \not= Y, \quad \text{jedi}(Y).
\]

\[
?- \text{help3}(X,Y).
\]

\[
?- \text{jedi}(X), \quad X \not= Y, \quad \text{jedi}(Y).
\]

\[
X=\text{luke}
\]

\[
?- \text{luke} \not= \text{luke}
\]

\[
\text{luke}=\text{luke}, !, \text{fail}
\]

\[
X=\text{yoda}
\]

\[
?- \text{yoda} \not= \text{yoda}
\]

\[
\text{yoda}=\text{yoda}, !, \text{fail}
\]
Using $\backslash=$

- In fact, given $X \backslash= Y$
  - will always fail if $X$ or $Y$ are not both instantiated

  $X \backslash= a$ // fails for $X=a$
  $a \backslash= Y$  // fails for $Y=a$
  $X \backslash= Y$  // fails for $X=Y$
Example Using $\not=$

- Example

\[
\begin{align*}
\text{jedi}(\text{luke}). \\
\text{jedi}(\text{yoda}). \\
\text{help2}(X, Y) & : \text{jedi}(X), \text{jedi}(Y), X \not= Y. \\
\text{help3}(X, Y) & : \text{jedi}(X), X \not= Y, \text{jedi}(Y). \\
\text{help4}(X, Y) & : X \not= Y, \text{jedi}(X), \text{jedi}(Y).
\end{align*}
\]

?- \text{help4}(X, \text{luke}).
false.

?- \text{help4}(\text{yoda}, \text{luke}).
true.
Quiz 6: What does this query return?

```
jedi(luke).
jedi(vader).
sith(vader).
true_jedi1(X) :- jedi(X), not(sith(X)).
?- true_jedi1(X).
```

A. X = luke
B. false
C. true
D. X = vader
Quiz 6: What does this query return?

jedi(luke).
jedi(vader).
sith(vader).
true_jedi1(X) :- jedi(X), not(sith(X)).
?- true_jedi1(X).

A. X = luke
B. false
C. true
D. X = vader
Quiz 7: What does this query return?

jedi(luke).
jedi(vader).
sith(vader).
true_jedi2(X) :- not(sith(X)), jedi(X).
?- true_jedi2(X)

A. X = vader
B. X = luke
C. false
D. true
Quiz 7: What does this query return?

\[
\begin{align*}
\text{jedi}(\text{luke}). \\
\text{jedi}(\text{vader}). \\
\text{sith}(\text{vader}). \\
\text{true}_{\text{jedi2}}(X) & : - \text{not}(\text{sith}(X)), \text{jedi}(X). \\
? & - \text{true}_{\text{jedi2}}(X) \\
\end{align*}
\]

A. \( X = \text{vader} \)  \\
B. \( X = \text{luke} \)  \\
C. \text{false}  \\
D. \text{true}
Prolog Summary

General purpose logic programming language
- Associated with AI, computational linguistics
- Also used for theorem proving, expert systems

Declarative programming
- Specify facts & relationships between facts (rules)
- Run program as queries over these specifications

Natural support for
- Searching within set of constraints
- Backtracking