Outline

- Prolog
  - Basic data types
  - Lists
  - Unification
  - Prolog search tree
  - Negation as failure
  - Some small but useful things in Prolog
  - Cut!

Execution of Prolog Programs

- Unification: variable bindings.
- Backward Chaining/
  Top-Down Reasoning/
  Goal-Directed Reasoning:
    Reduces a goal to one or more subgoals.
- Backtracking:
  Systematically searches for all possible solutions that can be obtained via unification and backchaining.

Cut!

- The goal “!”, pronounced “cut” always succeeds immediately.
- It has an important side effect: Once it is satisfied, it disallows either:
  - backtracking back over the cut, or
  - backtracking and applying a different clause of the same predicate to satisfy the present goal.
- You can think of satisfying cut as making a commitment both
  - to the variable bindings we’ve made during the application of this rule, and
  - to this particular rule itself.
**Cut!**

- The cut goal succeeds whenever it is the current goal, and the derivation tree is trimmed of all other choices on the way back to and including the point in the derivation tree where the cut was introduced into the sequence of goals.
- Cut tells us: "Do not pass back through this point when looking for alternative solutions. ! acts as a marker back beyond which Prolog will not go. All the choices made prior to the cut are set, and are treated as though they were the only possible choices.
- You can think of Cut as telling the interpreter: "Trust me – if you get this far in the clause, there’s no need to backtrack and try another choice for proving this goal, or to try another way of satisfying any of the subgoals that were already proved for this goal." The goal “!” pronounced “cut” always succeeds immediately.

**Use of Cut**

- Cut can be used to improve the efficiency of search by reducing Prolog’s search space. E.g.,

When two predicates are mutually exclusive.

\[
\begin{align*}
q(X) & :- \text{even}(X), a(X). \\
q(X) & :- \text{odd}(X), b(X).
\end{align*}
\]

With cut

\[
\begin{align*}
q(X) & :- \text{even}(X), !, a(X). \\
q(X) & :- \text{odd}(X), b(X).
\end{align*}
\]

**Use of Cut**

- Cut can remove unwanted answers.
- Consider the Family Database:

```
male(albert).
male(bob).
male(edward).
female(alice).
female(victoria).
parent(albert, edward).
parent(victoria, edward).
parent(albert, alice).
parent(victoria, alice).
parent(albert, bob).
parent(victoria, bob).
%- son(?X) iff X is a son
son(X):-parent(_,X), male(X).
```

**Use of Cut**

- Cut can remove unwanted answers, but ...
- Consider the Family Database:

```
male(albert).
male(bob).
male(edward).
female(alice).
female(victoria).
parent(albert, edward).
parent(victoria, edward).
parent(albert, alice).
parent(victoria, alice).
parent(albert, bob).
parent(victoria, bob).
%- son(?X) iff X is a son
son(X):-parent(_,X), male(X), !.
```
Use of Cut

- Cut is dangerous
  male(albert).
  male(bob).
  male(edward).
  female(alice).
  female(victoria).
  parent(albert,edward).
  parent(victoria,edward).
  parent(albert,alice).
  parent(victoria,alice).
  parent(albert,bob).
  parent(victoria,bob).
  % son(?X) iff X is a son
  son(X):-parent(_,X), !, male(X).

Think:
- any male is a potentially good answer, so we want to try all of them: can’t put “cut” after “male” in the same rule.
- if a male has 2 parents, we only want to list him once as the answer: want to put “cut” after “parent”.

Result:
  son(X):-male(X), isason(X).
  isason(X):-parent(_,X), !.

?- son(X).
 X = edward ;
 No.

% What really happened here?
% What if the order of the facts changes?

Use of Cut

What about sibling?
  sibling(X,Y):-parent(P,X),parent(P,Y).

?- sibling(alice,Asib).
 Asib = edward ;
 Asib = alice ;
 Asib = bob ;
 Asib = edward ;
 Asib = alice ;
 Asib = bob ;
 no.

Use of Cut

- Try to use cut:
  siblings(X,Y):-parent(P,X), !, parent(P,Y).

?- sibling(alice,Asib), !, parent(P,Y).
 Asib = edward ;
 Asib = alice ;
 Asib = bob ;
 No
 So far so good.

- But if we ask:
  siblings(X,Y):-parent(P,X), !, parent(P,Y).

?- sibling(Asib,alice).
 Asib = edward ;
 No
 Not so good.
Use of Cut

Think:
• any two people in the database are potentially good answers, so we want to try all of them: can’t put “cut” in a rule after X and/or Y is instantiated
• if 2 people share 2 parents, we only want to list them once as the answer: want to put “cut” after 2 “parent” rules.

Result:
sibling(X,Y):-person(X),person(Y), commonparent(X,Y).
person(X):-male(X).
person(X):-female(X).
commonparent(X,Y):=-parent(P,X),parent(P,Y),!.

?- sibling(alice,Asib).
   Asib = bob ;
   Asib = edward ;
   Asib = alice ;
   No

Use of Cut

Finally, we don’t want X to be a sibling on X.
sibling(X,Y):-not(X=Y), person(X),person(Y),
           commonparent(X,Y).
person(X):-male(X).
person(X):-female(X).
commonparent(X,Y):=-parent(P,X),parent(P,Y),!.

?- sibling(alice,Asib).
   Asib = bob ;
   Asib = edward ;
   Asib = alice ;
   No

What went wrong? How to fix? (see next page)

Use of Cut

When to use cut?
• Common uses of the !:
  – Tell the Prolog system that it has found the right rule for a particular goal: If you get this far, you have picked the correct rule for this goal.
    e.g., P(N,X):-odd(N), !, X is N+3. % assume that odd() is defined.
    P(N,X):-even(N), X is N / 2.
  – Tell the Prolog system to fail a particular goal immediately without trying for alternate solutions: If you get to here, you should stop trying to satisfy the goal.
    e.g., not(G):- G,!, fail.
    not(G).
  – Terminate the generation of alternative solutions: If you get to here, you have found the only solution to this problem, no point in looking for alternatives.
    e.g., factorial(N,X):- N>0, N1 is N-1, factorial(N1, X1), X is X1*N.
    factorial(0,1):-!.
**Prolog: cut ! - examples**

- **Double-step function:**
  - if X < 3 then Y = 0
  - if 3 =< X and X < 6 then Y = 2
  - if 6 =< X then Y = 4

  % In Prolog
  f(X,0) :- X < 3. % rule 1
  f(X,2) :- 3 =< X, X < 6. % rule 2
  f(X,4) :- 6 =< X. % rule 3

  %- f(4,Y).
  Y=2

  %- f(1,Y),2<Y.
  no

**Prolog: cut ! – example**

- **Double-step function - cont’d:**
  - if X < 3 then Y = 0
  - if 3 =< X and X < 6 then Y = 2
  - if 6 =< X then Y = 4

  What do we know about this function that Prolog doesn’t?

  % same relations with !
  f(X,0) :- X < 3,!. % rule 1
  f(X,2) :- 3 =< X, X < 6,!. % rule 2
  f(X,4) :- 6 =< X. % rule 3

  %- f(1,Y),2<Y.
  no

  In this example, we changed the procedural meaning of the program, but not the declarative meaning.

**Prolog: cut ! – example**

- **Double-step cont’d:**
  % same relations with !
  f(X,0) :- X < 3,!. % rule 1
  f(X,2) :- X < 6,!. % rule 2
  f(X,4). % rule 3

  | ?- f(5,Y). |
  | Y = 2 |

  Can we come up with a more efficient version?

  f(X,0) :- X < 3,!. % rule 1
  f(X,2) :- X < 6,!. % rule 2
  f(X,4). % rule 3
  ?- f(5,Y). % query
  Y = 2

  % What if we removed the cuts?
  f(X,0) :- X < 3. % rule 1
  f(X,2) :- X < 6. % rule 2
  f(X,4). % rule 3
  ?- f(1,Y). % query
  Y = 0; % right answer
  Y = 2; % wrong answer, why?
  Y = 4; % wrong answer, why?

  Here, we changed the procedural and also the declarative meaning.

**Prolog: cut ! classification**

- **Green cuts:**
  - Affect procedural meaning of program but has no effect on the declarative meaning.
  - Do not affect readability of programs.
  - Used mainly to avoid wasted computations.

- **Red cuts:**
  - Affect declarative meaning of program as well as procedural meaning.
  - Affect readability:
    - Similar to unconditional jump (goto) in imperative PLs
    - Used to avoid wasted computations and also introduce semantics
    - If not used cautiously, can affect result in arbitrary way.
cut ! - examples

• Member function:
  member(Element, [Element | _]).
  member(Element, [_ | Rest]) :- member(Element, Rest).

  % What’s the problem in the above function?
  member(Element, [Element | _]) :- !.

• Sum function:
  – Without !:
    sum_to(1,1).
    sum_to(N, Res) :- N1 is N-1, sum_to(N1,TRes), Res is TRes + N.
    ?- sum_to(1,X).
    X=1; %Infinite loop………………….
  – With !:
    sum_to(1,1) :- !.
    sum_to(N, Res) :- N1 is N-1, sum_to(N1,TRes), Res is TRes + N.
    ?- sum_to(1,X).
    X=1; 

cut ! – examples

• Bubble-sort without !:
  % Find 2 adjacent elements X and Y in List such that X > Y
  % and swap X and Y to get List1, then sort List1. If there is
  % no pair of adjacent X and Y where X > Y, we are done.
  bubblesort(List,Sorted):-swap(List,List1),
    bubblesort(List1,Sorted).

  swap([X,Y | Rest],[Y,X | Rest]) :- X > Y.
  % rule 1
  swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
  % rule 2
  ?- bubblesort([10,3,1],Res). %query
    Res = [1,3,10]; % more output ?

• Bubble-sort with !:
  bubblesort(List,Sorted):-swap(List,List1),!
    bubblesort(List1,Sorted).
  swap([X,Y | Rest],[Y,X | Rest]) :- X > Y.
  swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
  ?- bubblesort([10,3,1],X). %query
    X = [1,3,10]; 

Prolog: pros & cons

• Cons:
  – Horn clauses have limited expressive power
  – Closed world assumptions (anything not mentioned is false)
  – Ordering of clauses change the result
  – Because horn clause is the basic construct, you must program carefully to
    avoid infinite loops and incorrect negation.
    • There is no 1-solution-fits-all for these problems…

• Pros:
  – Pattern matching
  – Backtracking
  – Unification
  – Rules and goals are also data (dynamic programming).
  – The logical model is powerful

Die the death of a thousand cuts… * Richard O’Kaffe – Craft of Prolog
# Logic vs. Imperative Languages

<table>
<thead>
<tr>
<th></th>
<th>Pascal, C, Java etc.</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Progransections</strong></td>
<td>1. Types</td>
<td>1. Types (or domains)</td>
</tr>
<tr>
<td></td>
<td>2. Procedures</td>
<td>2. Predicates</td>
</tr>
<tr>
<td></td>
<td>3. Procedurebody</td>
<td>3. Clauses</td>
</tr>
<tr>
<td></td>
<td>4. Statements, proc-calls</td>
<td>4. predicate-calls</td>
</tr>
<tr>
<td></td>
<td>5. calculationsw/functions</td>
<td>5. calculationsw/functions</td>
</tr>
<tr>
<td></td>
<td>6. booleanexpressions, &lt;,=,&gt;, and, or, not ..., etc</td>
<td>6. booleanexpressions, &lt;,=,&gt;, and, or, not ..., etc</td>
</tr>
<tr>
<td><strong>Assignment</strong></td>
<td>Yes (x=x+1)</td>
<td>no</td>
</tr>
<tr>
<td><strong>Unification</strong></td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Parameterpassing</strong></td>
<td>simple</td>
<td>complex</td>
</tr>
<tr>
<td><strong>Patternmatching</strong></td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Top-down reasoning &amp; back-tracking</strong></td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>