**Today**

- Prolog Wrap Up (part 1)

**Assignments**

- HW-9, R-9 due Thurs
- HW-10, R-10 out
Watch out for backtracking ... 

A relation to zero-out an unwanted value ... 

• e.g., zero_out(3, [1,3,2,3,5], Xs) returns Xs = [1,0,2,0,5]

```
zero_out(_, [], []). % base case
zero_out(X, [X|Ys], [0|Zs]) :- % found value
    zero_out(X, Ys, Zs).
zero_out(X, [Y|Ys], [Y|Zs]) :- % didn’t find val
    X \== Y, zero_out(X, Ys, Zs).
```

Q: What happens if we leave out the X \== Y relation?

... HINT: Create a proof tree to see!
The Cut operator (!)

**Prolog and procedural programming**

- Consider this simple C++ function:

  ```cpp
  void writename(int x)
  {
    switch(x) {
      case 1: cout << "One"; break;
      case 2: cout << "Two"; break;
      case 3: cout << "Three"; break;
      default: cout << "Out of range";
    }
  }
  ```

Q: How would we implement this in Prolog?

- How not to do it ...

  ```prolog
  writename(1) :- write('One').
  writename(2) :- write('Two').
  writename(3) :- write('Three').
  writename(X) :- write('Out of range').
  ```

- An example of an “unwanted alternative”. Instead ...

  ```prolog
  writename(1) :- write('One').
  writename(2) :- write('Two').
  writename(3) :- write('Three').
  writename(X) :- X < 1, write('Out of range').
  writename(X) :- X > 3, write('Out of range').
  ```

- While this works, it isn’t very elegant ...
Instead, use the Prolog cut (!) operator

- Forces Prolog to stop searching for alternatives when an answer is found
  - A cut “burns backtracking bridges”
  - once executed, commits to the current solution
  - eliminates backtracking options

A very simple example

```
a :- b, c.
a :- d.
b.  
c.  
d. 
```

Q: What is the proof tree for `solve[a]`?

```
?- a.  
true ;  
true.  
```

- a is proved twice!
Now lets add a cut ...

\begin{verbatim}
  a :- b, !, c.
  a :- d.
  b.
  c.
  d.
\end{verbatim}

\begin{itemize}
  \item We end up only proving $a$ once ...
  \begin{itemize}
    \item The path to the cut is \texttt{``frozen''}
    \item We won't try different solutions for $b$ or $a$
    \item We will still try different solutions for $c$
  \end{itemize}
\end{itemize}
The effect of a cut is limited to the containing clause

\[
\begin{align*}
  a & :- b, !, c. \\
  a & :- d. \\
  p & :- q, a, r. \quad \% \text{"calls" a} \\
  p & :- s.
\end{align*}
\]

- For `solve[p]`
  - What happens if \( q \) and \( a \) succeed, but \( r \) fails?
  - Since \( a \) succeeded ... we don't backtrack **within** \( a \)
  - But we can backtrack **over** \( a \) ...
  - to try different \( q \)'s and the second clause of \( p \)

**Using cuts to help with writename**

\[
\begin{align*}
  \text{writename(1)} & :- !, \text{write('One')}. \\
  \text{writename(2)} & :- !, \text{write('Two')}. \\
  \text{writename(3)} & :- !, \text{write('Three')}. \\
  \text{writename(X)} & :- \text{write('Out of range')}.\]
\]

**Cuts can help with efficiency in Prolog**

\[
\begin{align*}
  \text{member}(X, [X|\_]). \\
  \text{member}(X, [\_|T]) & :- \text{member}(X, T).
\end{align*}
\]

Q: What is the proof tree for `solve[member(1, [1, 2])]`?

  - Note we check `member(1, [2])`
  - Even though we already computed the answer!

- We can avoid this using a cut:

\[
\begin{align*}
  \text{member}(X, [X|\_]) & :- !. \\
  \text{member}(X, [\_|T]) & :- \text{member}(X, T).
\end{align*}
\]
We could also rewrite this using negation (*):

\[
\text{member}(X, [X|\_]). \\
\text{member}(X, [H|T]) :- \text{\textbar}+ (X=H), \text{member}(X, T).
\]

where \text{\textbar}+ denotes “not” (negation)

Q: But can you see a possible problem using this or cuts?
   – What if we wanted to find all members of a list?

Q: For example, what will \text{member}(X, [1, 2]) return?
   – Only the first element!

(* In fact, negation in Prolog (\text{\textbar}+) is defined using a cut:

\[
\text{\textbar}+(P) :- \text{call}(P), \!, \text{fail}.
\]
\[
\text{\textbar}+(P).
\]