CPSC 312

Functional and Logic Programming

1 October 2015

accumulators: reverse

An accumulator is an added argument, that's not used as input or output and is treated as hidden, that holds a temporary value that's updated in between recursive calls.

```
reverse1([],[]).

reverse1([X|Xs],Zs) :- reverse1(Xs,Ys),append(Ys,
[X],Zs).


reverse2(Xs,Ys) :- reverse2(Xs, [], Ys).

reverse2([X|Xs],Acc,Ys) :- reverse2(Xs, [X|Acc], Ys).

reverse2([],Ys,Ys).
```

which implementation of reverse is more efficient?

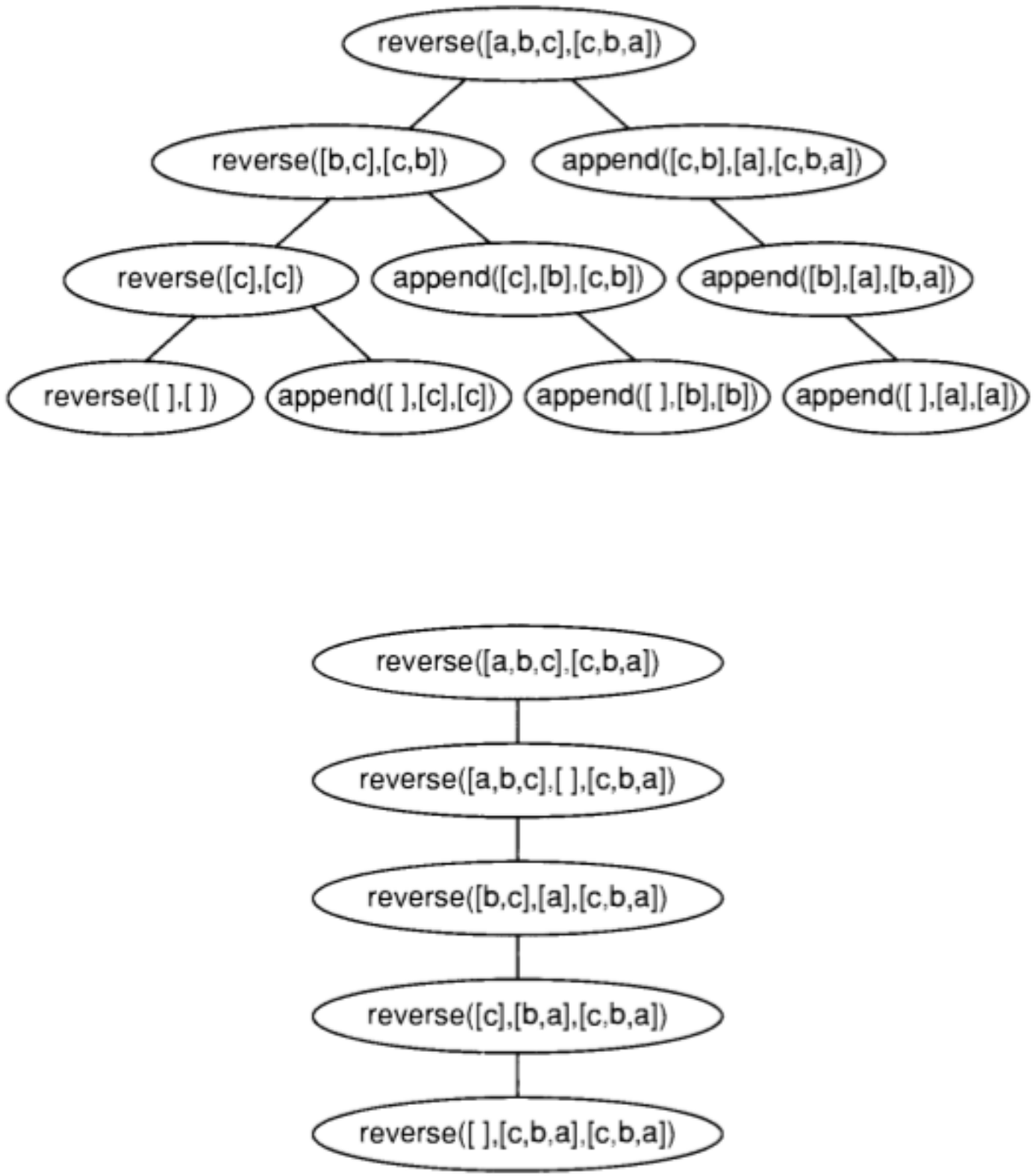


Figure 3.5 Proof trees for reversing a list

recap

- resolution and unification; the computational model of Prolog/logic programming
- procedural considerations; rule order, goal order, termination
- database programming; e.g. family tree, wiring, the maze
- lists and list processing
- recursion and how to write recursive procedures through induction
- arithmetic and stepping outside logic out of necessity

Cut's meaning

- The Cut, !, is a directive to resolution algorithm to prune its search trees by committing to the choices made so far.

- *"The goal succeeds and commits Prolog to all the choices made since the parent goal was unified with the head of the clause that cut occurs in."*

Cut !

- Affecting the procedural behaviour of a program. Reducing the search space by dynamically pruning the tree.

- The use of cut is controversial because most its uses only have a procedural meaning and disrupt the declarative semantics that we've been discussing so far (kind of like arithmetic).

remember this case?

question ☆ 113 views

Why does the query "parent(tom,bob)." return true AND false?

From the example program we were looking at in class today:

```
parent(tom, bob).
parent(pam, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```

If you run the query "parent(tom,bob).", it returns true AND false:

```
?- parent(tom,bob).
|    .
true ;
false.
```

If you run the query "parent(tom,liz).", it only returns true:

```
?- parent(tom,liz).
true.
```

remember this case?

question ☆ 113 views

Why does the query "parent(tom,bob)." return true AND false?

From the example program we were looking at in class today:

```
parent(tom, bob).
parent(pam, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```

If you run the query "parent(tom,bob).", it returns true AND false:

```
?- parent(tom,bob).
|    .
true ;
false.
```

If you run the query "parent(tom,liz).", it only returns true:

```
?- parent(tom,liz).
true.
```

```
?- parent(tom, bob), !.
true.
```

Cut !

- Cuts are usually placed inside a program (rather than in a query) and their purpose is to restore efficiency by avoiding backtracking.

- One way to look at is that the cut forces the interpreter to 'commit' to all the choices made so far and throw away the alternatives.

Cut !

- Cuts are usually placed inside a program (rather than in a query) and their purpose is to restore efficiency by avoiding backtracking.

- One way to look at is that the cut forces the interpreter to 'commit' to all the choices made so far and throw away the alternatives.

- Remember that any derivation chain in Prolog reflects a series of choices. Prolog remembers those choices:

Simple Prolog interpreter

The program (*P*):

```
sublist(X,Y) :-
 suffix(Z,Y),prefix(X,Z).

prefix(X,Y) :- append(X,Z,Y).

suffix(X,Y) :- append(Z,X,Y).

append([],X,X).
append([H1|T1],X,[H1|T2]) :-
 append(T1,X,T2).
```

remember this choice

G = sublist([b],[a,b,c]).

resolvent = append(Z',Z,[a,b,c]),prefix([b],Z)

$\square$ = {}

Initialize resolvent to goal *G* (the query)
while resolvent not empty *do*
  choose a goal *A* from the resolvent
  choose a (renamed) clause *A'* :- *B1*,…,*Bn*
    from program *P* such that *A* and *A'* unify
    with mgu $\square$
  (if no such goal and clause exist, exit
  the while loop)
  replace *A* by *B1*,…,*Bn* in the resolvent
  apply $\square$ to the resolvent and to *G*
If the resolvent is empty, *then* output *G*,
  else output *no*

Cut !

- Another way of looking at it is pruning the search tree.

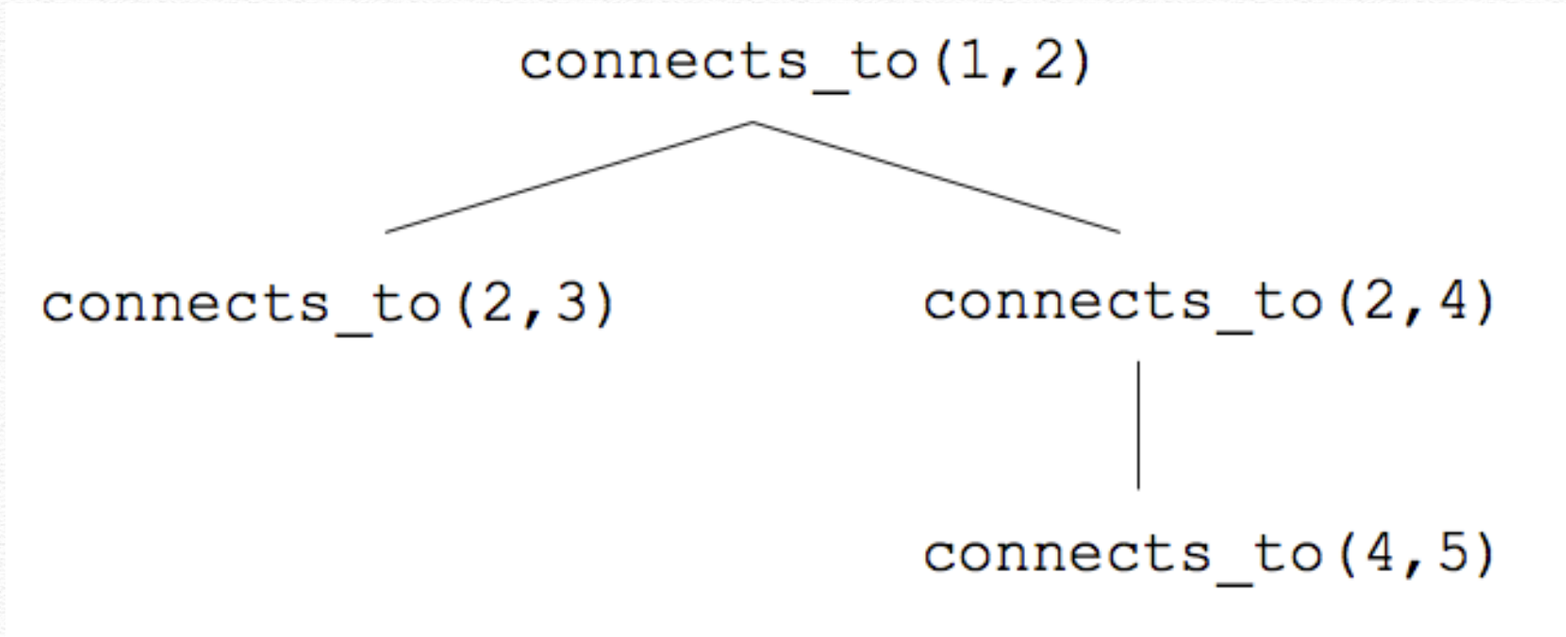
The Maze example

Consider:

```
connects_to(1,2)
connects_to(2,3)
connects_to(2,4)
connects_to(4,5)

path(X,Y):-connects_to(X,Y).
path(X,Y):- connects_to(X,Z),path(Z,Y).
```

The Maze example

```
?- path(1,5).
```



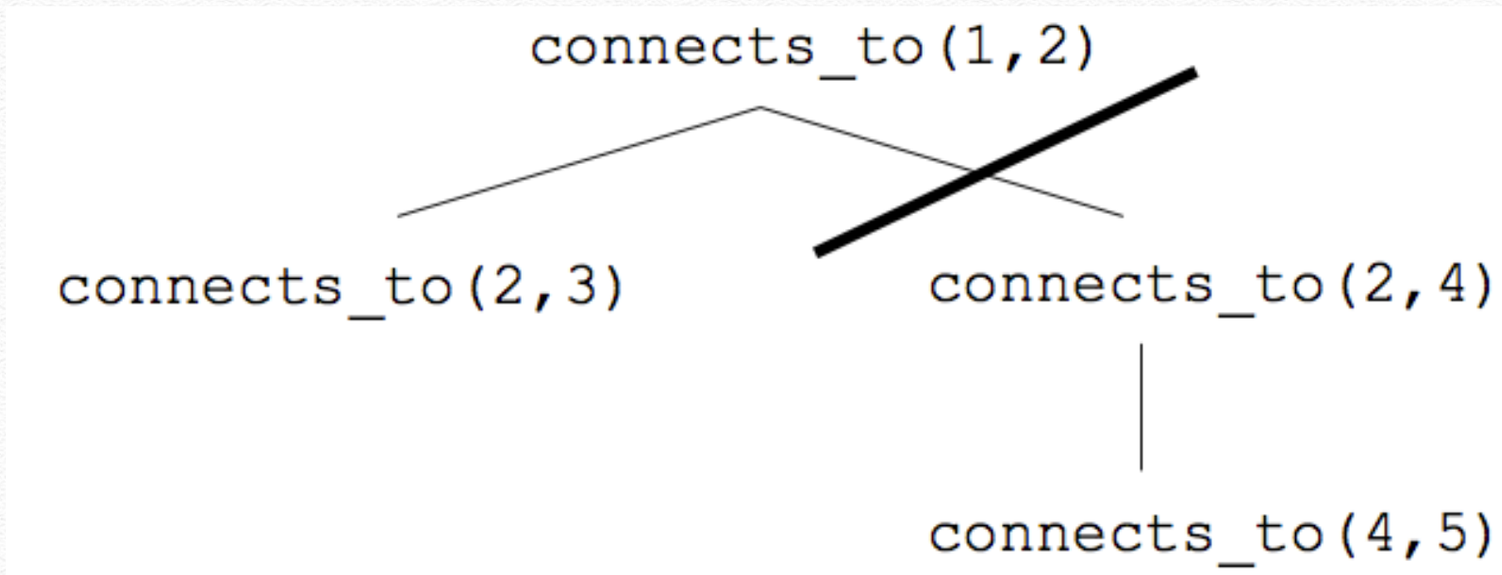

```
connects_to(1,2)
connects_to(2,3)
connects_to(2,4)
connects_to(4,5)
```

```
path(X,Y):- connects_to(X,Y).

path(X,Y):- connects_to(X,Z), path(Z,Y).
```

The Maze example

```
?- path(1,5).
```

```
connects_to(1,2)
connects_to(2,3)
connects_to(2,4)
connects_to(4,5)
```

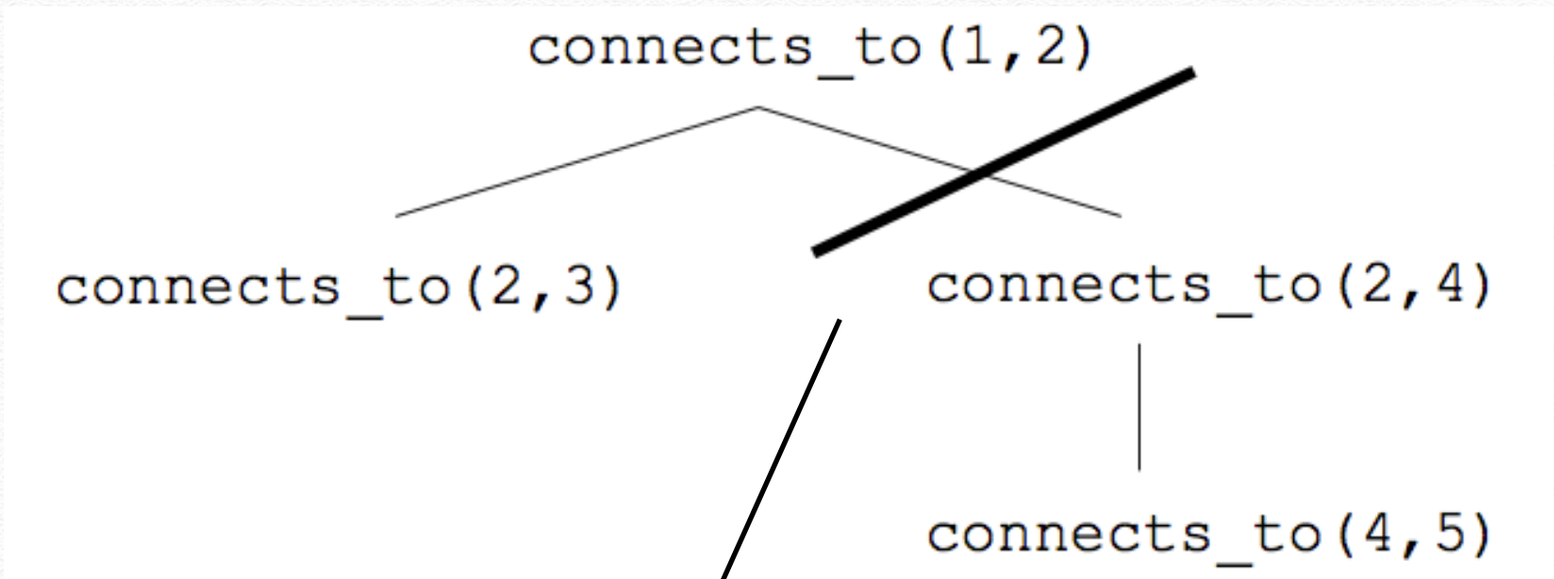



```
path(X,Y):- connects_to(X,Y).

path(X,Y):- connects_to(X,Z), path(Z,Y).
```

The Maze example

```
?- path(1,5).
```

```
connects_to(1,2)
connects_to(2,3)
connects_to(2,4)
connects_to(4,5)
```



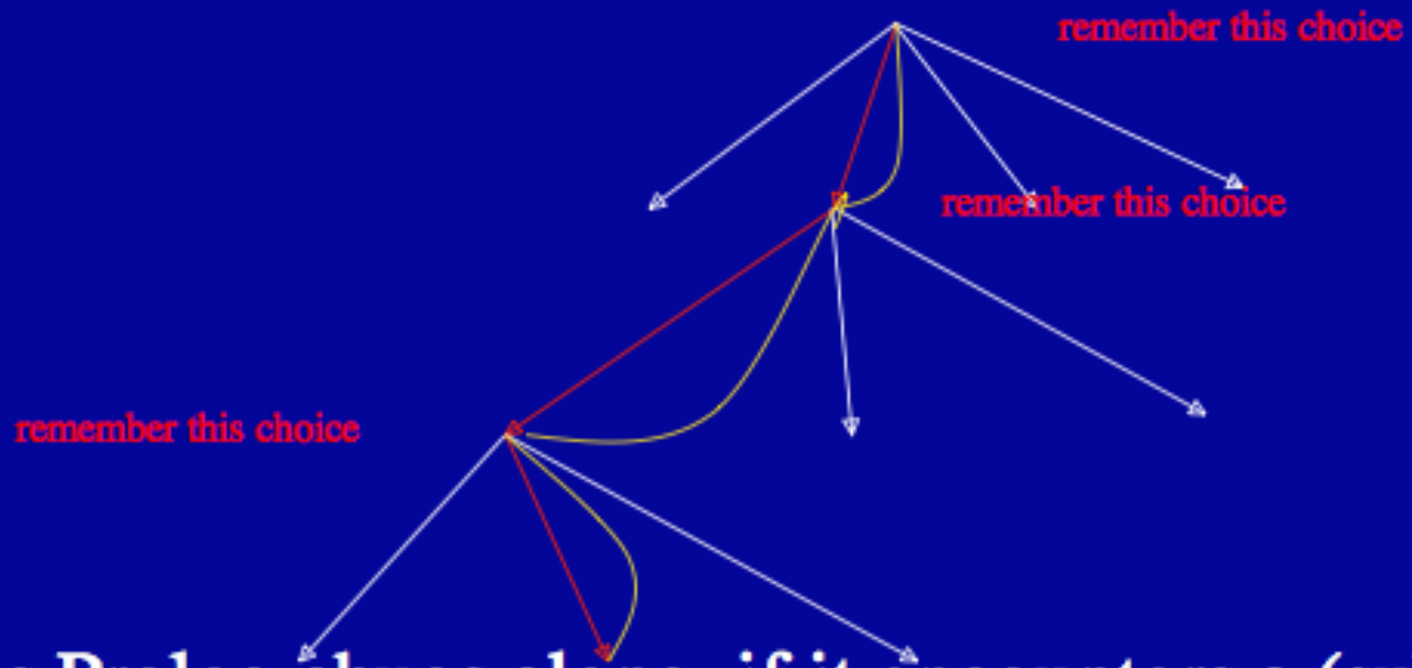

```
path(X,Y):- connects_to(X,Y).

path(X,Y):- connects_to(X,Z), !,path(Z,Y).
```

Or in still other different words...

Graphically, you can think of the derivation chain and alternate choices as looking like this:

Or in still other different words...

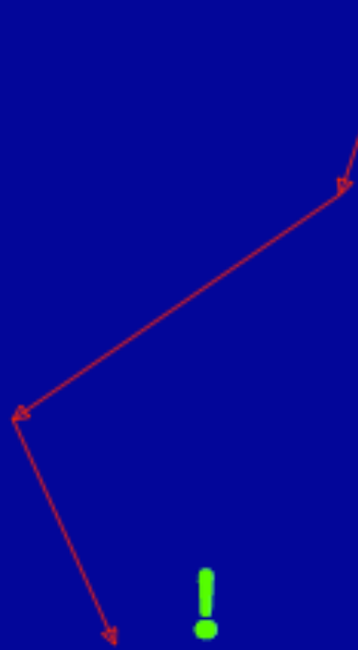
Graphically, you can think of the derivation chain and alternate choices as looking like this:



As Prolog chugs along, if it encounters a (sub)goal or conjunct from the resolvent that can't be proven, Prolog backtracks to the most recent choice and tries an alternative.

Or in still other different words...

When Prolog encounters a cut (!) symbol though, all those remembered choice points in the derivation are forgotten:



Or in still other different words...

When Prolog encounters a cut (!) symbol though, all those remembered choice points in the derivation are forgotten:

When Prolog processes a cut, it's like going through a one-way door -- you can go through the doorway, but once you've gone through, you can't go back. The cut commits Prolog to the derivation up to that point, with no do-overs.

The Cut

The cut symbol, denoted by !, is a pseudo-goal that you can use in your program to send that message to Prolog. It does the following things:

- It always succeeds
- It commits Prolog to all choices made between the time the 'parent goal' was invoked and the time the cut was encountered. All remaining alternatives between the parent goal and the cut are discarded.

More Formally..

- Consider the clause `H :- B1, B2, ..., Bm, !, ..., Bn`
- Assume you had a goal G that matched H. Then G is the 'parent goal'.
- When Prolog encounters the cut, it has already found some solution for B1,...,Bm. When the cut is executed, this current solution of B1,...,Bm is frozen and all possible alternatives are discarded.
- Also, the original goal G is now committed to this clause: any attempt to match G with the head of some other clause is precluded.

More Formally..

- Consider the clause `H :- B1, B2, ..., Bm, !, ..., Bn`
- The cut (!) prunes away all untried ways of solving goals `B1, B2, ..., Bm`

and

- The cut (!) prunes away all other ways of solving H, which is the head of the clause that unified with the parent goal
- So the solution if it is going to be found will only be found going forward (rightward) from the ! in this clause. If a goal fails to the right of this !, there will be no backtracking leftward past the !.

More Formally..

- Consider the clause `H :- B1, B2, ..., Bm, !, ..., Bn`
- The cut (!) prunes away all untried ways of solving goals `B1, B2, ..., Bm`

and

- The cut (!) prunes away all other ways of solving H, which is the head of the clause that unified with the parent goal
- So the solution if it is going to be found will only be found going forward (rightward) from the ! in this clause. If a goal fails to the right of this !, there will be no backtracking leftward past the !.
- Don't worry, it *is* hard to understand. The example will help clear things up.

Example: merging two sorted lists

```
mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :-H1 =:= H2,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2,
mymerge([H1|T1],T2,T3).
```

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)
```

```
mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :-H1 =:= H2,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2,
mymerge([H1|T1],T2,T3).
```

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2,
mymerge([H1|T1],T2,T3).
```

This is the first clause to unify with the goal.

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, mymerge(T1,[H2|
T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2, mymerge([H1|
T1],T2,T3).
```

As we can see, the three of the recursive clauses in this program are mutually exclusive. If a goal fits with one of them, it wouldn't match other ones. So once a match has been found we tell the interpreter not to backtrack and try other clauses.

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, mymerge(T1,
[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2,
mymerge([H1|T1],T2,T3).
```

At what point can you tell Prolog that it should commit to this clause and ignore the others? That is, where does the cut symbol go?

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, !,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2, mymerge([H1|
T1],T2,T3).
```

At what point can you tell Prolog that it should commit to this clause and ignore the others? That is, where does the cut symbol go? After the condition succeeds.

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, !,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2,
mymerge([H1|T1],T2,T3).
```

All these rules are mutually exclusive, so they could all have cut symbols. Where?

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1).

mymerge([],L2,L2).

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, !,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2, !,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2, !,
mymerge([H1|T1],T2,T3).
```

All these rules are mutually exclusive, so they could all have cut symbols. Where?

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1) :- !.

mymerge([],L2,L2) :- !.

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, !,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2, !,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2, !,
mymerge([H1|T1],T2,T3).
```

All these rules are mutually exclusive, so they could all have cut symbols. Where?

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1) :- !.

mymerge([],L2,L2) :- !.

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, !,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2, !,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2, !,
mymerge([H1|T1],T2,T3).
```

The cut symbols express the deterministic nature of mymerge: only one of the clauses can be used successfully for proving an applicable goal.

Example: merging two sorted lists

```
?- merge([1,3,5],[2,3],X)

mymerge(L1,[],L1) :- !.

mymerge([],L2,L2) :- !.

mymerge([H1|T1],[H2|T2],[H1|T3]) :- H1 < H2, !,
mymerge(T1,[H2|T2],T3).

mymerge([H1|T1],[H2|T2],[H1,H2|T3]) :- H1 =:= H2, !,
mymerge(T1,T2,T3).

mymerge([H1|T1],[H2|T2],[H2|T3]) :- H1 > H2, !,
mymerge([H1|T1],T2,T3).
```

The cut commits the computation to a single clause, once the computation has progressed enough to determine that this is the only clause to be used.

Summary to this point

- The cut predicate controls backtracking. It tells Prolog not to pass back through this point when looking for alternative solutions.

- The ! acts as a marker, back beyond which Prolog will not go. When it passes this point, all choices that it has made so far are locked in -- they are treated as though they are the only possible choices.

- Note that the cut always appears where a predicate can appear (never, for example, as an argument to a predicate). It is treated like any other predicate, and *it always succeeds*.

Summary to this point

The effect of the cut is as follows:

- Any variables which are bound to values at this point cannot take on other values.
- No other versions of predicates called before the cut will be considered.
- No other subsequent versions of the predicate at the head of the current rule will be considered.

Cut controversy

- Cut is one of the most controversial features of prolog while at the same time being one of the most useful ones.
- Let's face it. Prolog has some efficiency issues. Given the right problem domain, a little bit of insight, and a laissez-faire attitude about getting things done in reasonable time, we as programmers can construct elegant solutions to interesting problems relatively quickly, but when we stop to think about how the Prolog interpreter works, we see that our solutions may be doing lots of unnecessary work in the form of inferences that obviously don't need to be done.

Cut controversy

- Cut is a tool that if used correctly can restore (some) efficiency.
- Fact: you can't become a serious prolog programmer without learning to use the cut.
- At the same time, learning about the cut, *without a deep understanding of its use*, can turn you into a worse programmer than you were before cut appeared in your life.
- misunderstandings about the effects of a cut, i.e. which computations are cut and which ones are not, are a major source of bugs for inexperienced and experienced prolog programmers alike.

Merge example: expressing determinism

- Merge is deterministic in the sense that only one of the expressed cases can be true. The cases are mutually exclusive. The code has a very clear procedural meaning.

- It's safe to use cut here, because cutting alternative cases does not affect the meaning of the program.

Green cuts vs Red cuts

- As we've used it so far, the cut prunes the search tree which in turn reduces the amount of computation.

- In mymerge, cuts prune only computation paths that do not lead to new solutions. This kind of cut is called a **green cut**.

- Adding or removing green cuts does not change a program's meaning. They only change efficiency. All solutions to a given query will still be found.

Green cuts vs Red cuts

- Consider these two implementations of exchange (or interchange) sort from your book. One has no cuts, and the other has green cuts.

```
xsort(Xs,Xs) :- ordered(Xs).
xsort(Xs,Ys) :- append(As,[X,Y|Bs],Xs), X > Y,
append(As,[Y,X|Bs],Xs1), xsort(Xs1,Ys).

xsortc(Xs,Xs) :- ordered(Xs),!.
xsortc(Xs,Ys) :- append(As,[X,Y|Bs],Xs), X > Y, !,
append(As,[Y,X|Bs],Xs1), xsortc(Xs1,Ys).

ordered([]).
ordered([X]).
ordered([H1|[H2|T]]) :- H1 =< H2,ordered([H2|T]).
```

Green cuts vs Red cuts

Now run these tests at home. Use SWI-Prolog's time predicate to compare the differences in efficiency:

```
test1 :- xsort([12,11,10,9,8,7,6,5,4,3,2,1],
[1,2,3,4,5,6,7,8,9,10,11,12]).

test1c :- xsortc([12,11,10,9,8,7,6,5,4,3,2,1],
[1,2,3,4,5,6,7,8,9,10,11,12]).

test2 :- xsort([12,11,10,9,8,7,6,5,4,3,2,1],
[1,2,3,4,5,6,6,8,9,10,11,12]).

test2c :- xsortc([12,11,10,9,8,7,6,5,4,3,2,1],
[1,2,3,4,5,6,6,8,9,10,11,12]).

test3 :- xsort([6,5,4,3,2,1],[1,2,3,3,5,6]).

test3c :- xsortc([6,5,4,3,2,1],[1,2,3,3,5,6]).
```

Green cuts vs Red cuts

- But we can add cuts that do affect a program's meaning. For example, if we want to improve the efficiency of this:

```
member(X,[X|T]).
member(X,[H|T]) :- member(X,T).
```

  - we might do this:

```
member(X,[X|T]) :- !.
member(X,[H|T]) :- member(X,T).
```

Green cuts vs Red cuts

- But we can add cuts that do affect a program's meaning. For example, if we want to improve the efficiency of this:

```
member(X,[X|T]).
member(X,[H|T]) :- member(X,T).
```

- we might do this:

```
member(X,[X|T]) :- !.
member(X,[H|T]) :- member(X,T).
```

The result is that member is satisfied when the head of the first rule matches the goal. Prolog will not backtrack and try to find other solutions.

Green cuts vs Red cuts

```
member(X,[X|T]) :- !.
member(X,[H|T]) :- member(X,T).
```

```
?- member(X,[a,b,c]).
X=a.
```

Green cuts vs Red cuts

- A cut which affects what solutions will be found is called a red cut.
- **green** cut => green means **good**, go, small carbon footprint.
  red cut => red means **bad**, stop, Republican.
- Any cut that is not green is, by definition, red.
- Red cuts lose the correspondence between the nondeterministic meaning and the procedural meaning of a program. They change the solutions that the program will find.
- Use red cuts carefully, if at all.

Green cuts vs Red cuts

- That's not to say that red cuts can't be useful.
- Using red cuts we can write an if_then_else construct. How?

Green cuts vs Red cuts

- That's not to say that red cuts can't be useful.

- Using red cuts we can write an if_then_else construct. How?

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

Green cuts vs Red cuts

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

p=true.
?- if_then_else(p,write(1), write(2)).
1

p=false.
?- if_then_else(p,write(1), write(2)).
2
```

Green cuts vs Red cuts

If the red cut is removed:

```
if_then_else(P,Q,R) :- P, Q.
if_then_else(P,Q,R) :- R.
```

```
p=true.
?- if_then_else(p,write(1), write(2)),fail.
12
```

Upon backtracking the wrong answer will be tried and succeed.

Questions?

Questions?

- Read section 11.4 on Red cuts for a better understanding.

Negation as failure

- How can we implement negation in prolog? We've seen the predicate not/1 before, but how does it work?
- In a prolog program, you typically only express what *is* correct, not what isn't. Closed World Assumption ensures, what is not expressed or cannot be proven is false.
- So falseness is not something you can directly express. We haven't, for instance, been writing programs that contain lists of false facts.
- Therefore, negation is an implicit or derived condition modeled as 'failure to prove'.

not/1 or \+

The negation of a formula is true iff it cannot be proven true.

Thus if you cannot prove a(X), then not(a(X)) is considered true!

This definition in Prolog reflects the fact that we don’t know about our world is considered to be false.

not/1 or \+

```
not(P):- call(P), !, fail.
not(P).
```

`?- call(P)` forces an attempt to satisfy the goal P

This is also an if-then-else construct!

If call(P) succeeds, then fail, otherwise succeed.

not/1 or \+

```
not(P):- call(P), !, fail. not(P).
```

`?- call(P)` forces an attempt to satisfy the goal P

This is also an if-then-else construct!

If call(P) succeeds, then fail, otherwise succeed.

If P has variables, if any instances of those variables lead to satisfying P, then not/1 fails.

remember this example..?

How do we prove that fred is the father of everyone?

we can't we can only prove it's opposite:

```
?- not(father(fred,X)).
```

but this is not the true logical opposite.

Using negation

Unmarried student rule:

```
unmarried_student(X) :- not(married(X)),
student(X).
```

looks like the right code...

if I add:

```
student(bill).
married(joe).
```

would querying `?-unmarried_student(X)` produce `bill` as an output?

Using negation

```
unmarried_student(X) :-
not(married(X)), student(X).
```

```
student(bill).
married(joe).

?-unmarried_student(X).
false.
```

Using negation

```
unmarried_student(X) :- not(married(X)),
student(X).

student(bill).

married(joe).

?-unmarried_student(X).

false.
```

because married(X) unified with married(joe), therefore its negation fails.

how to fix it?

Using negation

```
unmarried_student(X) :- not(married(X)),
student(X).

student(bill).

married(joe).

?-unmarried_student(X).
false.
```

because married(X) unified with married(joe), therefore its negation fails.

how to fix it?

change the order of the clauses in the body of the rule.

Using Negation

```
unmarried_student(X) :- student,
not(married(X)).

student(bill).

married(joe).
```

- In SWI-Prolog, you need to use \+ instead of not, so

```
unmarried_student(X) :- student, \+
(married(X)).
```

what if I comment out married(joe) and keep the order as it was? would that work?

Using Negation

```
unmarried_student(X) :- student, not(married(X)).

student(bill).
married(joe).
```

- In SWI-Prolog, you need to use \+ instead of not, so:

```
unmarried_student(X) :- student, \+(married(X)).
```

what if I comment out `married(joe)` and keep the order as it was? would that work?

Not in SWI-Prolog:

```
ERROR: unmarried_student/1: Undefined procedure:
married/1
```

Using Negation

```
unmarried_student(X) :- student(X),
not(married(X)).
```

vs

```
unmarried_student(X) :-
not(married(X)), student(X).
```

Generalized lesson of this example: Negation does not or may not work as expected with non-ground goals.

Using Negation

- For the purpose of this class: learn very well the meaning of negation as failure and how its behaviour differs from what's expected of a logical negation (because it's not a logical negation).

- Remembering how not/1 (or \+) is implemented will help you reason about its behaviour and output.

- Study what's left of section 11.3 if you're interested in learning to effectively use negation.

Zebra Puzzle

1. There are five houses.
2. The Englishman lives in the red house.
3. The Spaniard owns the dog.
4. Coffee is drunk in the green house.
5. The Ukrainian drinks tea.
6. The green house is immediately to the right of the ivory house.
7. The Old Gold smoker owns snails.
8. Kools are smoked in the yellow house.
9. Milk is drunk in the middle house.
10. The Norwegian lives in the first house.
11. The man who smokes Chesterfields lives in the house next to the man with the fox.
12. Kools are smoked in the house next to the house where the horse is kept.
13. The Lucky Strike smoker drinks orange juice.
14. The Japanese smokes Parliaments.
15. The Norwegian lives next to the blue house.

Who drinks water?
Who owns the zebra?

Questions

Next Class

- We will see examples of Nondeterministic programming (ch. 14), introduce some useful extra-logical predicates (ch. 12) as well as some second-order predicates (ch. 16).

- The class after that, we will explores DCGs and Natural Language Processing in Prolog! (Ch. 19)