Programming Paradigms CSI2120 – Winter 2018

Jochen Lang

EECS, University of Ottawa

Canada

Logic Programming in Prolog

- **Backtracking with the Cut**
  - The “Cut”
  - Coding the existence of a single solution
    - no other rules should be tried
    - no backtracking across this point
  - Abandoning a goal with cut and fail
- **Data structures: Lists**
  - Basic list processing
  - Appending a list to another
  - Reading into a list

The “Cut” !

- **"cut" off some backtracking path**
  - Prolog will not try to re-satisfy certain goals.

- **Reasons for using the cut**
  - faster execution
  - more efficient use of memory (don't have to keep track of as many backtrack points)

- **Syntax: !**
  - goal that always succeeds and has a side effect (on the way backtracking works)

A Simple Example:

- **Goal: Removing branches from the search tree that are known not to produce a solution.**
- **Example: A cat is either a male or female cat. When one fact is proven, there is no point in searching for the opposite.**

```
cat(X) :- tomcat(X), !.
cat(X) :- female_cat(X), !.
```

  - The cut commits Prolog to the facts established
    - if the subgoal `tomcat(X)` succeeds, the cut succeeds and the cat rule in the first row succeeds. The backtracking path is now cut off and Prolog will not search to re-satisfy `tomcat(X)` or `cat(X)`.

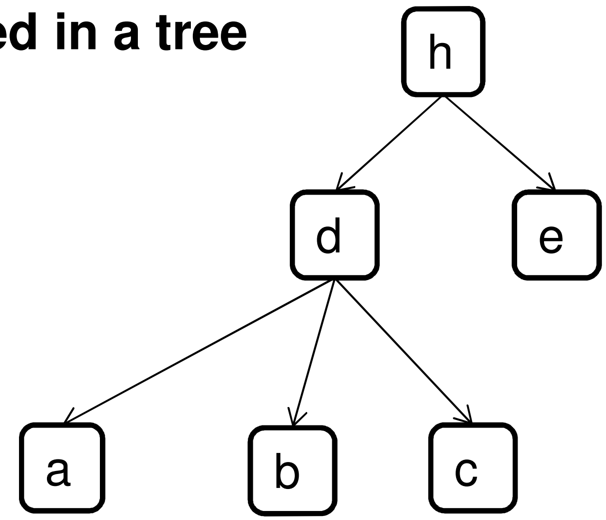
Effect of the Cut

- **When a cut is encountered as a goal, the system becomes committed to all choices made since the *parent goal was invoked*.**
  – This means the choice of which clause, and all the choices in the "sibling" goals up to the cut are fixed.
- **Therefore the Cut permits the elimination of branches in the search tree by**
  – eliminating other rules for the same goal
  – eliminating other choices for subgoals

Example: Search tree.

- **Rules and facts a,b,c,d,e,h arranged in a tree**

```
h(X) :- d(X).
h(X) :- X=e.
d(X) :- X=a.
d(X) :- X=b.
d(X) :- X=c.
a.
b.
c.
e.
```



- **Query initiating a tree traversal:**

```
?- h(X).
```

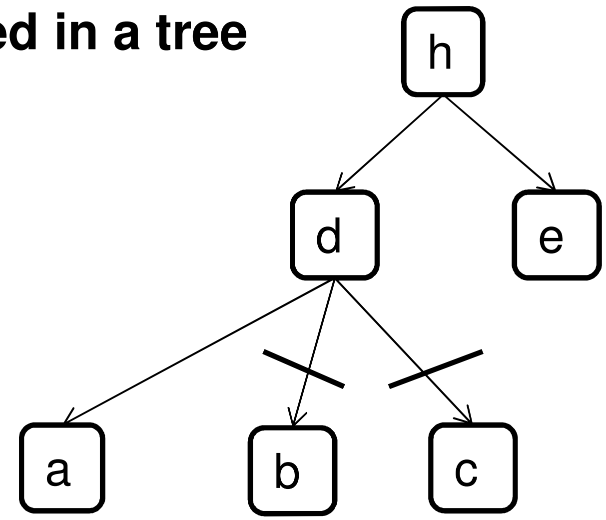
- **Produces the leave nodes (depth first search):**

  a b c e

Example: Search tree with cut branches

- **Rules and facts a,b,c,d,e,h arranged in a tree**

```
h(X) :- d(X).
h(X) :- X=e.
d(X) :- X=a,!.
d(X) :- X=b.
d(X) :- X=c.
a.
b.
c.
e.
```



- **Query initiating a tree traversal:**

```
?- h(X).
```

- **Produces only the leave nodes (depth first search):**

  a e

An Arithmetic Example: roots

- **A naive way to calculate the integer root of a number is to use a generator for integer numbers and see if the result succeeds.**
- **Generator**

```
int(0).
int(N) :- int(N1), N is N1+1.
```

- **Root predicate**

```
root(N,R) :- int(K), K*K>N,!, R is K-1.
```

  – Cut ensure that once $K^2 > N$, backtracking of the generator stops (`int`).

Red vs. Green Cuts

- **Red Cuts**
  - change the solution space (see the previous root example).
- **Green Cuts**
  - remove choices that wouldn't work anyway
  - efficiency gain
  - reduced memory footprint

Summary: Utility of the Cut

- Delete branches that will not lead to a solution
- Remove mutually exclusive cases in rules
- Reduce the number of solutions produced
- Make sure that certain programs (recursion) terminate
- Control of the proof procedure

If-then-else

- **Max function returning the larger of two numbers**

```
max(X,Y,X) :- X >= Y.
max(_,Y,Y).
```

- **This will produce surprising results, consider**

```
?- max(7,5,X).
   Call: (6) max(7, 5, _G1326) ? creep
   Call: (7) 7>=5 ? creep
   Exit: (7) 7>=5 ? creep
   Exit: (6) max(7, 5, 7) ? creep
X = 7 ;
   Redo: (6) max(7, 5, _G1326) ? creep
   Exit: (6) max(7, 5, 5) ? creep
X = 5.
```

If-then-else with the Cut

- **Solution: Use an explicit test**

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

- **Solution: Use the Cut for efficiency**

```
max(X,Y,X) :- X >= Y, !.
max(_,Y,Y).
```

- **Query**

```
?- max(7,5,X).
   Call: (6) max(7, 5, _G1326) ? creep
   Call: (7) 7>=5 ? creep
   Exit: (7) 7>=5 ? creep
   Exit: (6) max(7, 5, 7) ? creep
X = 7.
```

Predicate `fail`

- **The fail predicate always fails.**
- **Combined with the Cut we get negation**

```
is_false(P) :- P, !, fail.
is_false(P).
```

- **The clause P is either true then the goal is_false fails.**
  - Subgoal P is true, Cut is true cutting backtracking off, `fail` succeeds making the goal `is_false(P)` fail.
  - If subgoal P fails, the second rule succeeds.
- **The combination is called cut-fail.**

Built-in predicate not

- **Predicate `not` behaves as the previous example `is_false`.**

```
?- not(fail).
true.
?- not(X=1).
false.
?- not(X=1),X=0.
false.
```

- **Verifies the failure of a term.**
- **`not/1` should be used as a predicate with instantiated arguments for verification purposes.**
  - it does not generate solutions

Simple Example with Negation

```
happy_camper(X):- not(sad(X)), camper(X).
camper(joe).
sad(jane).
```

- **Queries**

```
?- happy_camper(joe).
true.
?- happy_camper(jane).
false.
?- happy_camper(X).
false.
```

Negation: not or \+

- **The negation predicate `not(F)` with F a term can also be written with the operator `\+F`**
  - Associative prefix operator
- **Negation by failure**
  - Prolog proofs the success of a goal `not(F)` by showing that it can not satisfy `F`.
    - Remember that Prolog may also fail to proof F for trivial reasons, e.g., it may simply missing facts in its database.
  - Prolog's proof strategy is referred to as closed world assumption.

Not Equals or Difference

- **We have seen the operator not equals before.**

```
X=\= Y.
```

- **Binary difference operator is the opposite of a successful unification of its' arguments.**
  – This operator is equivalent to the following definition

```
X =\= Y :- not(X = Y).
```

  – expressed with cut-fail

```
X =\= X :- !,fail.
X =\= Y.
```

Interval Predicate Again

- **Recall previous definition of a test for an interval and a separate definition of a generator for interval**

```
intervalTest(X,L,H):-  X>=L, X=<H
interval(X,X,H) :-     X=<H.
interval(X,L,H) :-     L<H, L1 is L+1,
                       interval(X,L1,H).
```

- **Definition using the Cut combining the two predicates**

```
interval(X,L,H) :-   number(X), number(L),
                     number(H), !, X>=L, X=<H.
interval(X,X,H) :-   number(X), number(H),
                     X=<H.
interval(X,L,H) :-   number(L),number(H),
                     L<H, L1 is L+1,
                     interval(X,L1,H).
```

Note the use of the built-in `number`

Lists

- **Common structure in Prolog**
  - A list holds objects (in the Prolog sense). Elements in a list may be list themselves.
- **Examples**
  - A list with three elements `[1, 2, 3]`
  - An empty list `[]`
  - Lists are processed in Prolog by referring to the head and tail of the list `[Head | Tail]`
  - Remember no typing, a valid list is `[1, 2, three]`
  - The tail of the list is always a list
  - The head of the list may consist of multiple elements `[1, 2, | Tail ]`

List Examples: Head and Tail

```
?- [1,2,three] = [X|Y].
X = 1,
Y = [2, three].
?- [1|[2,three]] = L.
L = [1, 2, three].
?- [1|[2,three]] = [X|Y].
X = 1,
Y = [2, three].
?- [1,2,three] = [X|[Y]].
false.
?- [1] = [X|Y].
X = 1,
Y = [].
?- [] = [X|Y].
false.
```

List Example

```
aList(X, Y, [X|Y]). % Simple predicate

?- aList(1, [2,3,4], L).
L= [1,2,3,4].

?- aList(X, Y, [1,2,3,4]).
X= 1,
Y= [2,3,4].

?- aList(1, [2,3,4], [1,2,3,4]).
true.
```

Basic List Processing

- **List membership**

```
listMember(X,[X|L]). % Searched for element is
                     % head - boundary case
listMember(X,[Y|L]) :-
   listMember(X,L).  % loop over list
```

- **Determining the Length of a List**

```
listLength([],0).    % empty list is length 0
listLength([X|L],N) :-
   listLength(L,NN), % loop over list
   N is NN+1.        % count
```

List Insertion with Permutations

```
listInsert(A,L,[A|L]).  % Position for A
listInsert(A,[X|L], [X|LL]) :-
    listInsert(A,L,LL). % Recursion with every
                        % level succeeding

?- listInsert(c, [a, b], L).
L = [c, a, b] ;
L = [a, c, b] ;
L = [a, b, c] ;
false.
```

Joining Lists

```
appendList([],Y,Y).
appendList([A|B],Y,[A|W]) :-
                appendList(B,Y,W).

?- appendList([1,2], [3,4], L).
L= [1,2,3,4].
?- appendList(X, [3,4], [1,2,3,4]).
X= [1,2]
?- appendList([1,2], [3,4], [1,2,3,4]).
true.
```

Example: List of Characters

- **Reading character from the console until line feed**

```
read_line(Line) :-
    get_char(C), read_line(C, Line).

read_line('\n', []) :- !.
read_line(C, [C | RestOfLine]) :-
  C \= '\n',
  get_char(NextC),
  read_line(NextC, RestOfLine).
```

- **Similar to SWI library predicate**

```
read_line_to_codes(user_input, ListChar).
```

Summary

- **Backtracking with the Cut**
  - Manipulating the search tree
  - Red and green cuts
  - Utility of the cut
- **Data structures: Lists**
  - Basic list processing
    - List membership, length of a list, append, permutation and insertion, reading a line