PROLOG NOTES #2

Structure Joining

```
append([],L,L).
append([H|T1],L2,[H|T3]):-append(T1,T2,T3).

?- append([a,b], [c,d], X).
X=[a,b,c,d].
?- append(Y, [c,d], [a,b,c,d]).
Y=[a,b].
```

Order of Predicates Matters

Not Good, will recurse forever:

```
isList([A|B]):-isList(B).
isList([]).
```

OK:

```
isList([]).
isList([A|B]):-isList(B).
```

Accumulators

- Ordinary Recursion: counting/result computation is done in back substitution, recusrsion usually works on a smaller input.
- Accumulator-Based Recursion: counting/result computation is done first and then passed to recursion.

Factorial Example

Ordinary Recursive:

```
factr(0,1).
factr(N,F):- J is N-1,factr(J,F1), F is N*F1.
```

Accumulators:

```
facti(N,F):-facti(0,1,N,F).
facti(N,F,N,F).
facti(I,Fi,N,F):- J is I+1, Fj is J*Fi, facti(J,Fj,N,F).
```

Length of List Example

Ordinary Recursive

```
listlen([],0).
listlen([H|T],N) :- listlen(T,Nt), N is Nt+1.
```

Accumulators

```
listlen(L,N) :- lenacc(L,0,N).
lenacc([],A,A).
lenacc([H|T],A,N) :- Ax is A+1, lenacc(T,Ax,N).
```

For initial argument [a,b,c,d] the arguments to lenacc as subsequently:

lenacc([a,b,c,d],0,N)
lenacc([b,c,d],1,N)
lenacc([c,d],2,N)
lenacc([d],3,N)
lenacc([],4,N)

At this pont lenacc([],A,A) is matched, therefore N is unified with 4 and goal is satisfied.

Fibonacci Example

Ordinary Recursive:

```
fib(0,1).
fib(1,1).
fib(N,F):-N1 is N-1, N2 is N-2,fib(N1,F1), fib(N2,F2), F is F1+F2.
```

?- fib(10,X).
X = 89 .
?- fib(Y,89).
ERROR: is/2: Arguments are not sufficiently instantiated
^ Exception: (8) _L136 is _G241-1 ? creep
?- fib(X,Y).
X = 0,
Y = 1 ;
X = 1,
Y = 1 ;
ERROR: is/2: Arguments are not sufficiently instantiated
^ Exception: (8) _L136 is _G235-1 ?

Accumulators:

```
fibt(0,1).
fibt(1,1).
fibt(N,F):-fibt(2,1,1,N,F).
```

```
fibt(N,Last2,Last1,N,F):-F is Last1+Last2.
fibt(I,Last2,Last1,N,F):-
        J is I+1,
        Fi is Last1+Last2,
        fibt(J,Last1,Fi,N,F).
```

?- fib(10,X).
X = 89 .
?- fibt(X,89).
X = 10 .
?- fibt(X,Y).
X = 0,
Y = 1 ;
X = 1,
Y = 1 ;
X = 2,
Y = 2 ;
X = 3,
Y = 3 ;
X = 4,
Y = 5 ;
X = 5,
Y = 8 ;
X = 6,
Y = 13 .

CUT !

Advantages:

a) Faster program execution – CUT = commit i.e. this is it, don't try alternatives if you backtrack to CUT. For example:
   foo :-a,b,c,!,d,e,f.
   if a,b,c succeed then we are commited to what d,e,f do – if they fail foo fails.
b) Less memory consumption – the information about alternatives is not remembered.

Disadvantages:

a) Programs are much harder to follow.
b) Some alternative results will not be found.

Common Use Cases:

a) Commit to the results obtained so far.
b) Fail the predicate totally – use: !,fail.
c) Terminate alternative solutions.

d) Prevent infinite loops – backtracking (even one that is not explicit and thus hard to anticipate) from the only solution may throw a predicate into an infinite loop.

Example of Commit

```
sum_to(1,1):-!.
sum_to(N,Res):- N1 is N-1, sum_to(N1,Res1), Res is Res1+N.
```

Better yet, catch all grounding conditions to be more robust (behave well on bad input):

```
sum_to(N,1):-N=<1,!.
sum_to(N,Res):- N1 is N-1, sum_to(N1,Res1), Res is Res1+N.
```

Fail the Predicate Totally Example

If sth2 is true then don't try anything else:

```
sth1 :-sth2,!,fail.
```

For example:

```
sibling(A,B):-A=B,!,fail.
sibling(A,B):-parentOf(A,P),parentOf(B,P).
```

Prevention of Infinite Loops

The below program for do_sth would go into an infinite loop if the re was no CUT in sum_to:

```
do_sth :-sum_to(3,X),sth_bad.
sth_bad:-fail.
```

Example of Efficiency

The predicate \+X will succeed if X fails.

In the below example B will be evaluated twice if it fails:

```
A:-B,C.
A:\+B,D.
```

In the below example B will be evaluated only once:

```
A:-B,!,C.
A:-D.
```

Problems with CUT

No alternatives:

```
append([],L,L):-!.
Append([H|T1],B,[H|T2]):-append(T1,B,T2).
?-append(X,Y,[a,b,c]).
X=[], Y=[a,b,c].
```

Using CUT to Handle Exceptions

Exceptions do not commit:

number_of_parents(adam,0).
number_of_parents(eve,0).
number_of_parents(_,2).
?- number_of_parents(dam,2).
True.

Exceptions do commit:

number_of_parents(adam,N):-!,N=0.
number_of_parents(eve,N):-!,N=0.
number_of_parents(_,2).
?- number_of_parents(dam,2).
No.

Not

The NOT predicate is built in but it is logically equivalent to:

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

The uninstantiated variables do not get instantiated inside NOT. Not acts as an existential quantifier thus if not instantiated, then X inside NOT is not the same as the X outside of it:

?- not(not(member(X,[a,b,c]))),write(X).
_G421
true.