CPSC 312

Functional and Logic Programming

24 September 2015

Assignment 1

- Due next Tuesday at the beginning of the class. I'll be collecting hard copies at the start of the class.
- You can also hand it in today (after the class), or slide it under my office door (ICCS 187) anytime <u>before noon next Tuesday.</u>

Office Hours

- Instructor: (Sara Sagaii) sarams@cs.ubc.ca
  - Tuesdays 10-11am
  - Thursdays 10:30-11:30am
  - ICCS 187
- Rui Ge:
  - Wednesdays 10-12am
  - Table 1 at DLC
- Susanne Bradley:
  - Fridays 11:30-1:30
  - Table 1 at DLC
- Khurram Ali:
  - Mondays 3:30-5:30
  - Place: TBA

Questions

Recursion: review

- A recursive procedure consists of three parts:
- The base case or termination condition. Usually the first thing done upon entering a recursive procedure
- The reduction step -- the operation that moves the computation closer to the termination condition
- The recursive procedure calling itself

Recursion: the prolog view

- The underlying logic is the same, what's different is the higher level thinking.
- remember: Prolog procedures don't return the value you're looking for, they just return yes or no.
- So instead of saying "compute the factorial of 4 and return it to me"...
- you really want to write a procedure that will prove, e.g. that 24 is the factorial of 4. we talked about induction as a way to carry out this proof.
- ...then when you want to know what the factorial of 4 is, you just leave a variable in the query where you would have put the 24.

Recursion: arithmetics

```
natural(0).
natural(s(X)) :- natural(X).

plus(0,X,X).
plus(s(X),y,s(Z)) :-
plus(X,Y,Z).
```

Recursion: arithmetics

```
natural(0).
natural(s(X)) :- natural(X).

plus(0,X,X).
plus(s(X),y,s(Z)) :- plus(X,Y,Z).

times(0,X,0).
times(s(X),Y,Z) :-
times(X,Y,P),plus(P,Y,Z).
```

Recursion: arithmetics

```
natural(0).
natural(s(X)) :- natural(X).

plus(0,X,X).
plus(s(X),y,s(Z)) :- plus(X,Y,Z).

times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,P),plus(P,Y,Z).
```

in Prolog: `times(X,Y,Z) :-`
`succ(Xa,X),times(Xa,Y,P),plus(P,Y,Z).`

Recursion: arithmetics

```
exp(X,N,Y)?
```

Recursion: arithmetics

`exp(X,N,Y)?`

Use Induction.

What is the base case?

```
exp(X,0,1).
exp(0,N,0).
```

Recursion: arithmetics

`exp(X,N,Y)?`

Use Induction.

What is the base case?

```
exp(X,0,1).
exp(0,N,0).
```

if exp(X,N,Y), what can be said about exp with N+1 or s(N)?

Recursion: arithmetics

```
factorial(N, F)?
```

Recursion: arithmetics

`factorial(N, F)?` meaing?

Recursion: arithmetics

`factorial(N, F)?` meaing?

Use induction.

what is the base case?

Recursion: arithmetics

`factorial(N, F)?` meaing?

Use induction.

what is the base case?

```
factorial(0,s(0)).
```

Recursion: arithmetics

`factorial(N, F)?` meaing?

Use induction.

what is the base case?

`factorial(0,s(0)).`

if factorial(N,F) is true, what is factorial of N+1, or s(N)?

Recursion: arithmetics

`factorial(N, F)?` meaing?

Use induction.

what is the base case?

```
factorial(0,s(0)).
```

if factorial(N,F) is true, what is factorial of N+1, or s(N)?

```
factorial(s(N),F) :-
factorial(N,Fn),times(s(N),Fn,F).
```

Recursion: arithmetics

`factorial(N, F)?` meaing?

Use induction.

what is the base case?

```
factorial(0,s(0)).
```

if factorial(N,F) is true, what is factorial of N+1, or s(N)?

```
factorial(s(N),F) :-
factorial(N,Fn),times(s(N),Fn,F).
```

rewrite in Prolog: `factorial(M,F) :-`

```
succ(N,M),factorial(N,Fn),times(M,Fn,F).
```

List Processing

```
list([]).
list([_|T]) :- list(T).

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

last(X,[X]).
last(X,[_|T]) :- last(X,T).
```

List Processing

adding a new item to a list?

List Processing

adding a new item to a list?

as simple as `add(X,L,[X|L]).`

List Processing

adding a new item to a list?

as simple as `add(X,L,[X|L]).`
or even: `[X|L]`

List Processing

adding a new item to a list?

as simple as `add(X,L,[X|L]).`
or even: `[X|L]`

what's the meaning of this add function?

List Processing

adding a new item to a list?

as simple as `add(X,L,[X|L]).`
or even: `[X|L]`

what is the meaning of this add function?

what if we want to add X to the end?

List Processing

adding a new item to a list?

as simple as `add(X,L,[X|L]).`
or even: `[X|L]`

what if we want to add it to the end?

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

List Processing: delete

List Processing: delete

```
del(X,L,Lx)
```

List Processing: delete

`del(X,L,Lx)` meaning?

List Processing: delete

`del(X,L,Lx)` meaning?

base case?

List Processing: delete

`del(X,L,Lx)` meaning?

base case? `del(X,[X|Xs],Xs).`

List Processing: delete

`del(X,L,Lx)` meaning?

base case? `del(X,[X|Xs],Xs).`

what is the induction step?

List Processing: delete

```
?- del(a,[b,c,a,e],X).
```

List Processing: delete

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

List Processing: delete

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

?- del(a,X,[b,c,e]).
```

List Processing: delete

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

?- del(a,X,[b,c,e]).
X = [a, b, c, e] ;
X = [b, a, c, e] ;
X = [b, c, a, e] ;
X = [b, c, e, a] ;
false.
```

List Processing: delete

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

?- del(a,X,[b,c,e]).
X = [a, b, c, e] ;
X = [b, a, c, e] ;
X = [b, c, a, e] ;
X = [b, c, e, a] ;
false.
```

what else does this result resemble?

List Processing: delete

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

?- del(a,X,[b,c,e]).
X = [a, b, c, e] ;
X = [b, a, c, e] ;
X = [b, c, a, e] ;
X = [b, c, e, a] ;
false.
```

what else does this result resemble? how about insert a in any place in the list?

List Processing:delete

```
add(X,L,[X|L]).
```

List Processing:delete

```
add(X,L,[X|L]).
```

can we use add for deleting?

List Processing:delete

```
add(X,L,[X|L]).
```

can we use add for deleting?

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

List Processing:delete

```
add(X,L,[X|L]).
```

can we use add for deleting?

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

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

List Processing:delete

```
add(X,L,[X|L]).
```

can we use add for deleting?

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

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

yes, but only from the head...

List Processing:delete

```
add(X,L,[X|L]).
```

can we use add for deleting?

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

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

yes, but only from the head...and that's fine because the meaning of add is also add to the head.

List Processing:delete

```
add(X,L,[X|L]).
```

can we use add for deleting?

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

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

yes, but only from the head...and that's fine because the meaning of add is also add to the head.

from textbook (on how to blend declarative and procedural thinking p. 65): "Construct a program with a given use in mind; then consider if the alternative uses make declarative sense".

List Processing:delete

what about the meaning of delete? Is the delete function we wrote declaratively sound?

List Processing:delete

what about the meaning of delete? Is the delete function we wrote declaratively sound?

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

List Processing:delete

what about the meaning of delete? Is the delete function we wrote declaratively sound?

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

X = [b, c, a] ;

X = [b, a, c] ;

false.
```

List Processing:delete

what about the meaning of delete? Is the delete function we wrote declaratively sound?

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

X = [b, c, a] ;

X = [b, a, c] ;

false.
```

How can we fix that?

List Processing

How can we rewrite this to make it remove all occurrences?

```
del(X,[X|Xs],Xs).

del(X,[H|Xs],[H|Y]) :-
del(X,Xs,Y).
```

List Processing

How can we rewrite this to make it remove all occurrences?

```
del(X,[X|Xs],Y):- del(X,Xs,Y).
del(X,[H|Xs],[H|Y]) :-
del(X,Xs,Y).
```

List Processing

How can we rewrite this to make it remove all occurrences?

```
del(X,[],[]).
del(X,[X|Xs],Y):- del(X,Xs,Y).
del(X,[H|Xs],[H|Y]) :-
del(X,Xs,Y).
```

List Processing

The two versions of del are semantically or declaratively different.

List Processing

The two versions of del are semantically or declaratively different. Here is an example how:

```
member(X,L) :-
  del(X,L,_).
%only the first version of del
```

List Processing

The two versions of del are semantically or declaratively different. Here is an example how:

```
member(X,L) :-
  del(X,L,_).
%only the first version of del
```

which version gives a more *natural* definition of delete?

List Processing

There is another problem with the second version of del as well..

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

List Processing

There is another problem with the second version of del as well..

```
?- del2(a,[b,a,c,a],X).
X = [b, c] ;
X = [b, c, a] ;
X = [b, a, c] ;
X = [b, a, c, a] ;
false.
```

List Processing

How can we fix that?

```
del(X,[],[]).

del(X,[X|Xs],Y):- del(X,Xs,Y).

del(X,[H|Xs],[H|Y]) :-
del(X,Xs,Y).
```

List Processing

How can we rewrite this to make it remove all occurrences?

```
del(X,[],[]).

del(X,[X|Xs],Y):- del(X,Xs,Y).

del(X,[H|Xs],[H|Y]) :- X \= H
del(X,Xs,Y).
```

this is an issue of avoiding backtracking..we will talk about it more when we discuss the *cut* (!)

List Processing

On your own:

- length of list? `length([a,b,c]),3)` or `length([a,b,c],s(s(s(0)))).`
- `equal_length(L1,L2)` ? (without using length)
- insert at a given point? `insert(X,L,N,Lx).`
- `reverse([a,b,c],[c,b,a]).`

List Processing: append

- Something we do frequently in list processing is join one list to another, giving one larger list. This is often called append or conc (for concatenate) or sometimes + +.

- declarative perspective: we need to write a procedure which proves that a given list is the result of appending one list to another. For example, we want our procedure to prove

```
append([a,b,c],[d,e],[a,b,c,d,e])
```

List Processing: append

- So how do you prove this?

```
append([a,b,c],[d,e],[a,b,c,d,e])
```

Induction.

Prove `append([b,c],[d,e],[b,c,d,e])`

Prove `append([c],[d,e],[c,d,e])`

Prove `append([],[d,e],[d,e]).` That's easy.

```
append([],X,X).
```

What's the induction step?

List Processing: append

- The previous slides give us these relationships:
- `append( [a,b,c], [d,e], [a,b,c,d,e])`
- `append( [b,c], [d,e], [b,c,d,e])`
- `append( [c], [d,e], [c,d,e])`
- `append( [], [d,e], [d,e])`

List Processing: append

- The previous slides give us these relationships:
- `append( [a,b,c], [d,e], [a,b,c,d,e])`
- `append( [b,c], [d,e], [b,c,d,e])`
- `append( [c], [d,e], [c,d,e])`
- `append( [], [d,e], [d,e])`
- Can you generalize this? Look for patterns

List Processing: append

- The previous slides give us these relationships:
- `append( [a,b,c], [d,e], [a,b,c,d,e])`
- `append( [b,c], [d,e], [b,c,d,e])`
- `append( [c], [d,e], [c,d,e])`
- `append( [], [d,e], [d,e])`
- Can you generalize this? Look for patterns
- `append([H1|T1], X , [H2|T2] ) <- append( T1 , X , T2 )`

List Processing: append

- The previous slides give us these relationships:

- `append( [a,b,c], [d,e], [a,b,c,d,e])`
- `append( [b,c], [d,e], [b,c,d,e])`
- `append( [c], [d,e], [c,d,e])`
- `append( [], [d,e], [d,e])`

- Can you generalize this? Look for patterns

- `append([H|T1], X , [H|T2] ) <- append( T1 , X , T2 )`

List Processing: append

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

List Processing: append

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

Append turns out to be a powerful tool whose utility extends beyond just joining lists together. Append can be used to split lists into component pieces by partially specifying the nature of the components.

List Processing: append

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

Append turns out to be a powerful tool whose utility extends beyond just joining lists together. Append can be used to split lists into component pieces by partially specifying the nature of the components.

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

List Processing: append

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

Append turns out to be a powerful tool whose utility extends beyond just joining lists together. Append can be used to split lists into component pieces by partially specifying the nature of the components.

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

List Processing: append

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

Append turns out to be a powerful tool whose utility extends beyond just joining lists together. Append can be used to split lists into component pieces by partially specifying the nature of the components.

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

All the legal ways to split a list..

List Processing

```
?- prefix([a,b],[a,b,c]).
```

List Processing

```
?- prefix([a,b],[a,b,c]).
```

Try this: list X is a prefix of list Y if Y can be split into two other lists, where X is the first list of those two other lists.

List Processing

```
?- prefix([a,b],[a,b,c]).
```

Try this: list X is a prefix of list Y if Y can be split into two other lists, where X is the first list of those two other lists.

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

List Processing

```
?- prefix([a,b],[a,b,c]).
```

Try this: list X is a prefix of list Y if Y can be split into two other lists, where X is the first list of those two other lists.

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

List Processing

```
?- prefix([a,b],[a,b,c]).
```

Try this: list X is a prefix of list Y if Y can be split into two other lists, where X is the first list of those two other lists.

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

The other list?

List Processing

```
?- prefix([a,b],[a,b,c]).
```

Try this: list X is a prefix of list Y if Y can be split into two other lists, where X is the first list of those two other lists.

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

The other list? It's just any other list that's not X or Y.

List Processing

```
?- prefix([a,b],[a,b,c]).
```

Try this: list X is a prefix of list Y if Y can be split into two other lists, where X is the first list of those two other lists.

```
prefix(X,Y) :- append(X,_,Y).
```

The other list? It's just any other list that's not X or Y.

Exercise

```
?- suffix([c,d],[a,b,c,d]).

?- sublist([a,b],[c,d,a,b,e]).
```

Exercise

1. How do you write a program using append that splits this list into a list of months before September and months after and including Septermber?

```
[jan,feb,mar,apr,may,jun,jul,aug,sep,
oct,nov,dec]
```

2. divide a list into two equal length lists?

```
dividelist([a,b,c,d,e],[a,b,c],[d,e])
```

3. Rewrite `member` with append.

Assignment 2

- will be announced later today or tomorrow..

infinite lists

- lists can be potentially infinite. what's the use of that? well, firstly, it's good for representing incomplete data.
- remember `member(b,X)`? X is an incomplete list. All we know is that it has at least b as a member. it could have other members too, we just don't know.
- Remember we said before that prolog operates under the *Closed World Assumption*; meaning if something has not been expressed and cannot be deduced then it's definitely wrong, i.e. false. The answer is not maybe. Incomplete list is a way to circumvent that rule and add data on the fly.
- try `?- member(a, [b,c|T]), last([b,c|T],X).`

Questions

Next Class

- We have covered all chapters from 1 to 7 (with the exception of 5). you can finish reading them now if you haven't.
- Next Week: Arithmetic, Cuts and Negation (Ch. 8 & 11)