Logic Programming

ICS 1015

247696M

Edward Fleri Soler

Contents

1) Recursive Functions

a) DescSort

```
/*function to determine max item in list*/
max(X,[X]).
max(H,[H|T]):- max(M,T), H > M.
max(Max,[H|T]):- max(Max,T), Max >= H.

/*finds position of X in list*/
f(X,[X],1).
f(X,[X|_],1).

/*recursive call*/
f(X,[_|T],C):- f(X,T,Temp), C is Temp + 1.

/*deletes item at position X in list*/
d(_,[],_):- write('Incorrect Input!'), nl.
d(1,[_|T],T).

/*recursive call*/
d(X,[H|T],[H|R]):- Q is X-1, d(Q,T,R).

/*sorts list in descending order*/
descSort([],_).
descSort([X],[X]).

/*recursive call*/
descSort(L,[Max|T]):- max(Max,L), f(Max,L,N), d(N,L,Temp),
descSort(Temp,T).
```

b) Occur

```
/*base case*/
occur(_,[],0).

/*increments count and recursively calls itself when item is at front of
list*/
occur(X,[X|T],A):- occur(X,T,Temp), A is Temp+1.

/*recursively calls itself when item is not at from of list*/
occur(X,[_|T],A):- occur(X,T,A).
```

c) CountDel

```
/*counts number of occurences of X in list and deletes all occurences*/

/*base case*/
countDel(_,[],[],0).

/*deletes occurence and increments count*/
countDel(X,[X|T],R1,R2):- countDel(X,T,R1,Temp), R2 is Temp + 1.

/*recursive call*/
countDel(X,[H|T],[H|R1],R2):- countDel(X,T,R1,R2).
```

d) CollCount

```
/*counts the number of occurances of X in a list, returns the count and
removes all X from the list*/
countDel(_,[],[],0).
countDel(X,[X|T],R1,R2):- countDel(X,T,R1,Temp), R2 is Temp + 1.

/*recursive calll*/
countDel(X,[H|T],[H|R1],R2):- countDel(X,T,R1,R2).

/*Outputs tuples containing items in the list together with their
occurence count*/
collCount([],[]).
collCount([H],[(H,1)]).

/*recursive call/call to countDel*/
collCount([H|T],[(H,N)|T2]):- countDel(H,[H|T],Temp,N),
collCount(Temp,T2).
```

e) RemDup

```
/*function to find the max item in a list*/
max(X,[X]).
max(H,[H|T]):- max(M,T), H > M.
max(Max,[H|T]):- max(Max,T), Max >= H.

/*function to delete all occurences of an item in a list*/
d(_,[],[]).
d(X,[X|T],F):- d(X,T,F).
d(X,[H|T],[H|R]):- d(X,T,R).

/*function to remove duplicate items and output list in descending
order*/
remDup([],[]).
remDup([_],[]).
remDup(L,[Max|T]):- max(Max,L), d(Max,L,Temp), remDup(Temp,T).
```

2) Word Functions

a) Textify

```
/*function to replace occurence of C1,C2 and C3 with NC in a list*/
textify([C1,C2,C3],_,[X],[X]).

/*base case*/
textify([C1,C2,C3],NC,[C1,C2,C3],[NC]).

/*recursive calls when text to be replaced is at the front (and not) of
the list*/
textify([C1,C2,C3],NC,[C1,C2,C3|T],[NC|R]):- textify([C1,C2,C3],NC,T,R).
textify(L,NC,[H|T],[H|L2]):- textify(L,NC,T,L2).
```

b) WordTextify

```
/*function which searches for the occurence of X in the list and
replaces it with Y/*

/*base case*/
wordTextify(_,_,[],[]).

/*recursive call when X is at the front (and not) of the list*/
wordTextify(X,Y,[X|T],[Y|R]):- wordTextify(X,Y,T,R).
wordTextify(X,Y,[Z|T],[Z|R]):- wordTextify(X,Y,T,R).
```

c) FullText

```
/*word replacement in list*/
textify([C1,C2,C3],_,[X],[X]).

/*base case*/
textify([C1,C2,C3],NC,[C1,C2,C3],[NC]).

/*recursive call with word at front of list*/
textify([C1,C2,C3],NC,[C1,C2,C3|T],[NC|R]):- textify([C1,C2,C3],NC,T,R).

/*recursive call with word not at front of list*/
textify(L,NC,[H|T],[H|L2]):- textify(L,NC,T,L2).

/*base case for fullText*/
fullText([A,B,C],[NC],L,Y):- textify([A,B,C],NC,L,Y).

/*tests for first 3 letters in WordListToReplace and then recursively
calls itself and textify function*/
fullText([A,B,C|T],[H1|T1],L,Y):- textify([A,B,C],H1,L,Temp),
fullText(T,T1,Temp,Y).
```

3) Definite Clause Grammars

a) DCG

i) Implementation of DCG using difference lists

```
s(X,Z):- np(X,Y),vp(Y,Z). /*initial difference list sentence structure*/

np(X,Z):- det(X,Y), noun(Y,Z). /*nounphrase split up into det and noun*/
np(X,Z):- np(X,Y), pp(Y,Z).  /*nounphrase split up into nounphrase and
preposition phase*/

vp(X,Z):- verb(X,Y), np(Y,Z).  /*verbphrase split up into verb and np*/
vp(W,Z):- verb(W,X), np(X,Y), pp(Y,Z).  /*verbphrase split up into verb,
np and preposition phrase*/

pp(X,Z):- prep(X,Y), np(Y,Z). /*preposition phase split up into
preposition and noun phrase*/

/*list of predetermined words*/

det([the|X],X).
det([a|X],X).
noun([waiter|Y],Y).
noun([meal|Y],Y).
noun([table|Y],Y).
noun([day|Y],Y).
verb([brought|W],W).
prep([to|Z],Z).
prep([of|Z],Z).
```

ii) Query to check all possible sentences

```
| ?- s(A,[]).
```

Output:

```
A = [the,waiter,brought,the,waiter] ;

A = [the,waiter,brought,the,meal] ;

A = [the,waiter,brought,the,table] ;

A = [the,waiter,brought,the,day] ;

A = [the,waiter,brought,a,waiter] ;

A = [the,waiter,brought,a,meal] ;

A = [the,waiter,brought,a,table] ;

A = [the,waiter,brought,a,day] ;

A = [the,waiter,brought,the,waiter,to,the,waiter] ;

A = [the,waiter,brought,the,waiter,to,the,meal] ;

A = [the,waiter,brought,the,waiter,to,the,table] ;

A = [the,waiter,brought,the,waiter,to,the,day] ;

A = [the,waiter,brought,the,waiter,to,a,waiter] ;

A = [the,waiter,brought,the,waiter,to,a,meal] ;

A = [the,waiter,brought,the,waiter,to,a,table] ;

A = [the,waiter,brought,the,waiter,to,a,day] ;

A = [the,waiter,brought,the,waiter,to,the,waiter,to,the,waiter] ;

A = [the,waiter,brought,the,waiter,to,the,waiter,to,the,meal] ;

A = [the,waiter,brought,the,waiter,to,the,waiter,to,the,table] ;

A = [the,waiter,brought,the,waiter,to,the,waiter,to,the,day] ;

A = [the,waiter,brought,the,waiter,to,the,waiter,to,a,waiter] ;
```

```
A = [the,waiter,brought,the,waiter,to,the,waiter,to,a,meal] ;

A = [the,waiter,brought,the,waiter,to,the,waiter,to,a,table] ;

A = [the,waiter,brought,the,waiter,to,the,waiter,to,a,day] ;.....
```

iii) DCG syntactics

This DCG is syntactually correct. This is because although the output may not make sense in spoken English, it is abiding by the syntactic rules of the language. This means that although the sentence might make no sense at all, the structure of the sentence is syntactically correct.

b) DCG query testing

i) “The waiter brought the meal to the table”

```
| ?- s([the,waiter,brought,the,meal,to,the,table],[]).

Output: yes
```

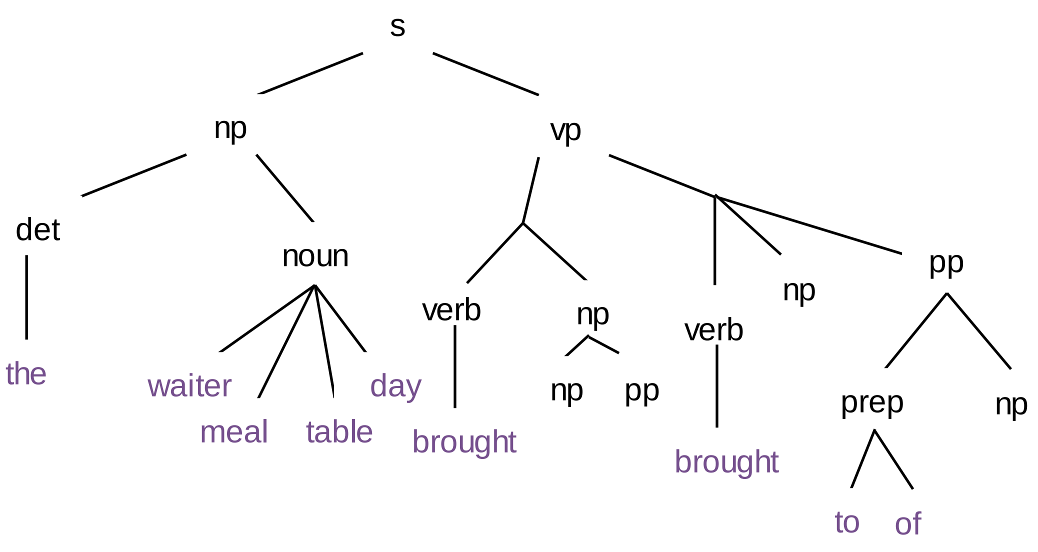
ii) “a meal brought by the waiter”

```
| ?- s([a,meal,brought,by,the,waiter],[]).

Output: no
```

iii) “the waiter brought the meal of the day”

```
| ?- s([the,waiter,brought,the,meal,of,the,day],[]).

Output: yes
```

c) Parse Tree for DCG



4) Binary Trees

a) Binary Tree Insert

```
/*test for emptyTree keyword*/
insert(I,emptyTree,bTree(nil,I,nil)).

/*test whether item to be inserted is greater or larger than root node*/
insert(I,bTree(nil,X,nil),bTree(bTree(nil,I,nil),X,nil)):- I=<X.
insert(I,bTree(nil,X,nil),bTree(nil,X,bTree(nil,I,nil))):- I>X.

/*handles trees with more nodes than a single root node*/
insert(I,bTree(W,X,nil),bTree(W,X,bTree(nil,I,nil))):- I>X.
insert(I,bTree(nil,X,Y),bTree(bTree(nil,I,nil),X,Y)):- I=<X.
insert(I,bTree(W,X,Z),bTree(R,X,Z)):- I=<X, insert(I,W,R).
insert(I,bTree(Z,X,Y),bTree(Z,X,R)):- I>X, insert(I,Y,R).
```

b) Binary Tree PreOrder

```
/*function to append an item to a list*/
app([],X,[X]).
app([H|T],X,[H|R]):-app(T,X,R).

/*function to merge two lists together*/
listMerge(L,[],L).
listMerge(L,[H|T],R):- app(L,H,R1), listMerge(R1,T,R).

/*handles case of empty list*/
preOrder(emptyTree,[]).

/*handling of various cases of tree structures*/
preOrder(bTree(nil,Y,nil),[Y]).
preOrder(bTree(nil,Y,Z),[Y|R]):- preOrder(Z,R).
preOrder(bTree(X,Y,nil),[Y|R]):- preOrder(X,R).

/*left and right subtrees handles seperately and then merged together*/
preOrder(bTree(X,Y,Z),([Y|R])):-preOrder(X,R1), preOrder(Z,R2),
listMerge(R1,R2,R).
```

c) Binary Tree Search

```
/*function for appending item to a list*/
app([],X,[X]).
app([H|T],X,[H|R]):-app(T,X,R).

/*function to merge two lists together*/
listMerge(L,[],L).

/*call to app aswell as recursive call*/
listMerge(L,[H|T],R):- app(L,H,R1), listMerge(R1,T,R).

/*preOrder binary tree sorting algorithm with different tree cases*/
preOrder(emptyTree,[]).
preOrder(bTree(nil,Y,nil),[Y]).
preOrder(bTree(nil,Y,Z),[Y|R]):- preOrder(Z,R).
preOrder(bTree(X,Y,nil),[Y|R]):- preOrder(X,R).
preOrder(bTree(X,Y,Z),([Y|R])):-preOrder(X,R1), preOrder(Z,R2),
listMerge(R1,R2,R).
```

```
/*search for a single element in a binary tree*/
elementSearch(R,[R|L]).

/*recursive call*/
elementSearch(R,[_|L]):- elementSearch(R,L).

/*final search function*/
search(L,R):- preOrder(L,X), elementSearch(R,X).
```