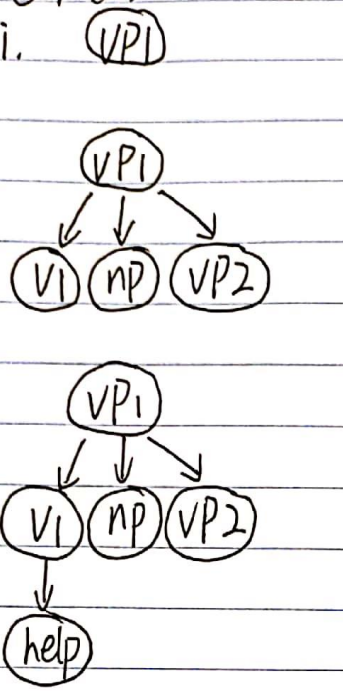
Can Zhao 19324118 Chosen Q1 and Q2.

Q1 (a)

i.



ii.

WORDS: help her eat it	STACK: VP1
WORDS: help her eat it	STACK: V1 NP VP2
WORDS: her eat it	STACK: NP VP2
WORDS: eat it	STACK: VP2
WORDS: eat it	STACK: V2 NP
WORDS: it	STACK: NP
WORDS:	STACK:

Q1 (b)

i.

WORDS: help her eat it.	STACK: VP
WORDS: help her eat it	STACK: V1 np VP.
WORDS: her eat it	STACK: np VP
WORDS: eat it	STACK: VP
WORDS: eat it	STACK: V1 np VP.

so, this version without backtracking will fail to parse the input.

ii.

WORDS: help her eat it	STACK: VP
WORDS: help her eat it	STACK: V1 np VP
WORDS: her eat it	STACK: np VP
WORDS: eat it	STACK: VP
WORDS: eat it	STACK: V1 np VP
BACKTRACKING to use rule: VP --> V1, np, VP	
WORDS: eat it	STACK: VP
WORDS: eat it	STACK: V2 np
WORDS: it	STACK: np
WORDS:	STACK:

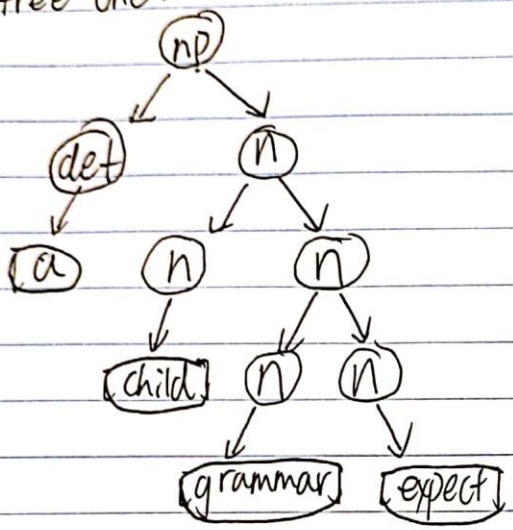
commentary (a) line

commentary for BACTRACKING:
when the parser is at * a dead end at (d) line.
the backtrack stack records 2 choices made so far.
0: STACK: VP (VP --> V1, np, VP)
1: STACK: ~~VP~~ VP (VP --> V1, np, VP)
so, it backs up to the most recent recorded choice point 1 and works from that.

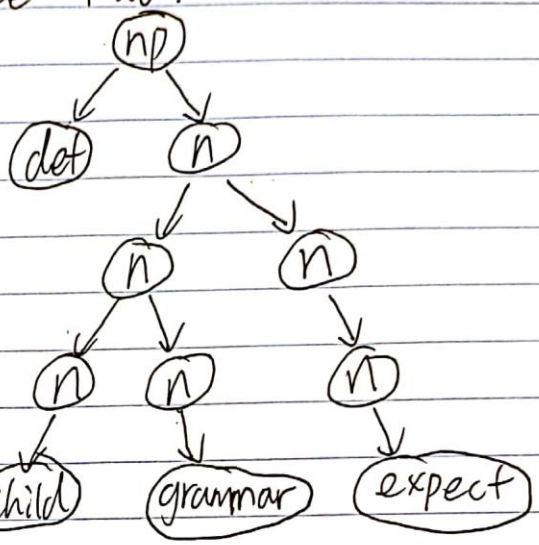
Q1(c).

i.

tree one:



tree two:



ii.

2^n

Since each acge has two different analyses.

so. let acge also has two different analyses.

Thus, the number of different analyses for (let acge)^n eat it is

$\underbrace{2 \times 2 \times 2 - \cdots \times 2}_{n} = 2^n$

There has 2^n different analyses for (let acge)^n. eat it.

Q1(d)

	Start Position			
length	0 (help)	1 (her)	2 (eat)	3 (it)
1	V1 V2	NP	V2	NP
2	VP		VP	
3		NP-VP		
4	VP			

Q1(e). Here are three main reasons:

0. To implement CKY table, we only need to store the categories which are from the rules. Storing the rest parse trees doesn't make much sense.
1. If we store the whole trees, it will be less space efficient.
2. When we combined two cells to an upper cell (like get cell (0,2) from the combination of cell(0,1) and cell(1,1)) It will be more complex to implement in tree structure than combined directly from categories.

Q1(f)

	Start Position			
length	0 (help)	1 (her)	2 (eat)	3 (it)
1	V1 VP/NP VP V2 VP/NP	NP	V2 VP/NP	NP
2	VP/VP VP		VP	
3				
4	VP			

(Q2 (a)

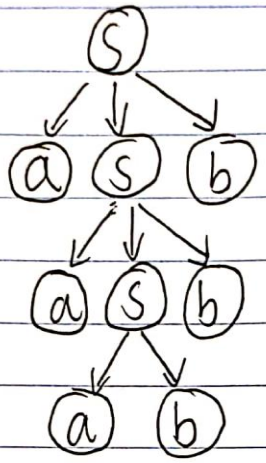
grammar:

S --> [a], S, [b]

S --> [a], [b]

initial (S)

tree:



See the rest on next pages

Q2 (b)

```
Tree * Passify (Tree *T1) {
  // create a new *tree : passify_tree.
  Tree * passify_tree = new Tree (T1 -> mother);

  // add DET, N, [which] to passify_tree in order
  passify_tree -> dtrs.push_back (T1 -> dtrs [0]);   // add DET
  passify_tree -> dtrs.push_back (T1 -> dtrs [1]);   // add N
  passify_tree -> dtrs.push_back (T1 -> dtrs [2]);   // add [which]

  // create a new *tree for TV-PASS
  Category tvPass ("TV-PASS");
  Tree* TV_PASS = new Tree (tvPass);

  // add [was], TV to TV-PASS in order
  Category c_was ("[was]");
  Tree * was = new Tree (c_was);
  TV_PASS -> dtrs.push_back (was);          // add [was] to TV_PASS
  TV_PASS -> dtrs.push_back (T1 -> dtrs[4]); // add TV to TV_PASS

  // add TV-PASS to passify_tree.
  passify_tree -> dtrs.push_back (TV_PASS);

  // add [by] to passify_tree.
  Category c_by ("[by]");
  Tree * by = new Tree (c_by);
  passify_tree -> dtrs.push_back (by);

  // add NP to passify_tree.
  passify_tree -> dtrs.push_back (T1 -> dtrs [3]);

  return passify_tree;
}
```

(Q2 cc)

added rules:

np --> det, n, relpro, s/np

relpro --> [that]

s/np --> vp

s/np --> np, vp/np.

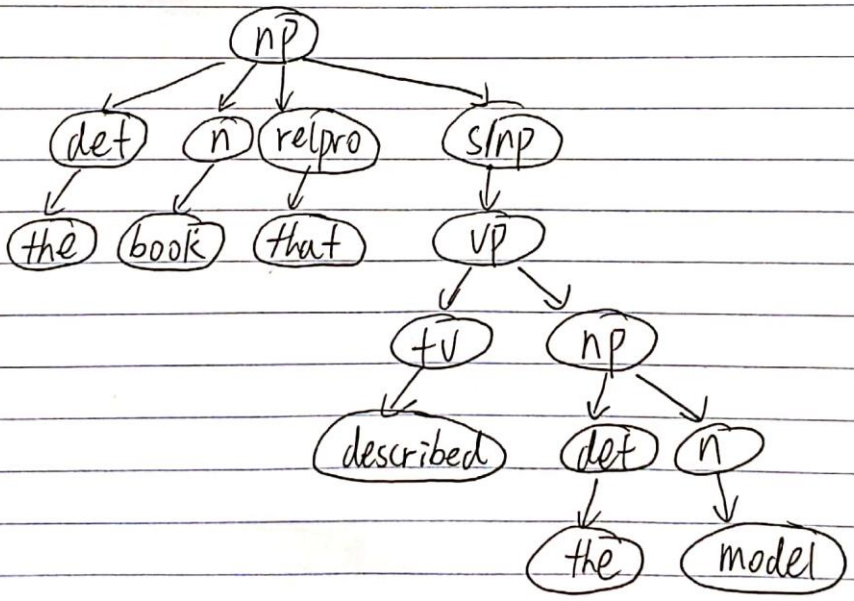
vp/np --> tv

vp/np --> [put], np, [on]

vp/np --> [forced], np, [to], vp/np.

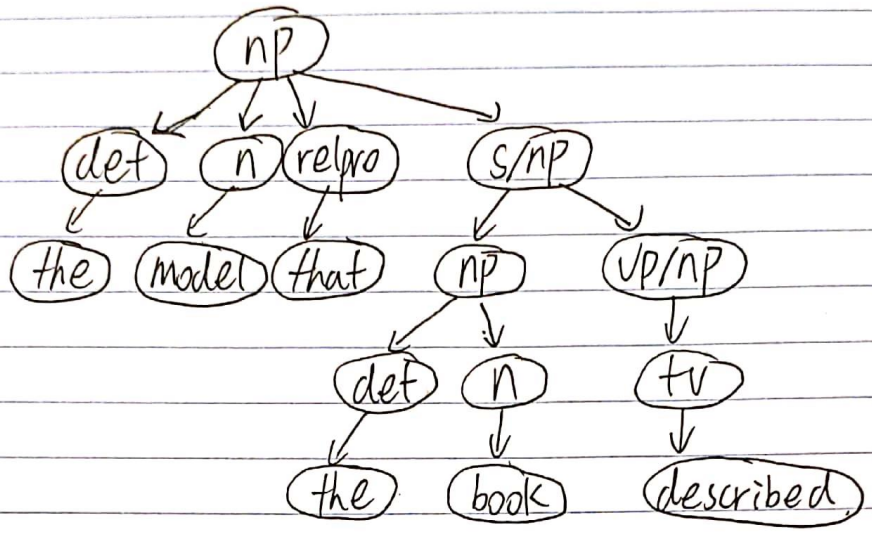
~~trees~~ trees:

input: the book that described the model

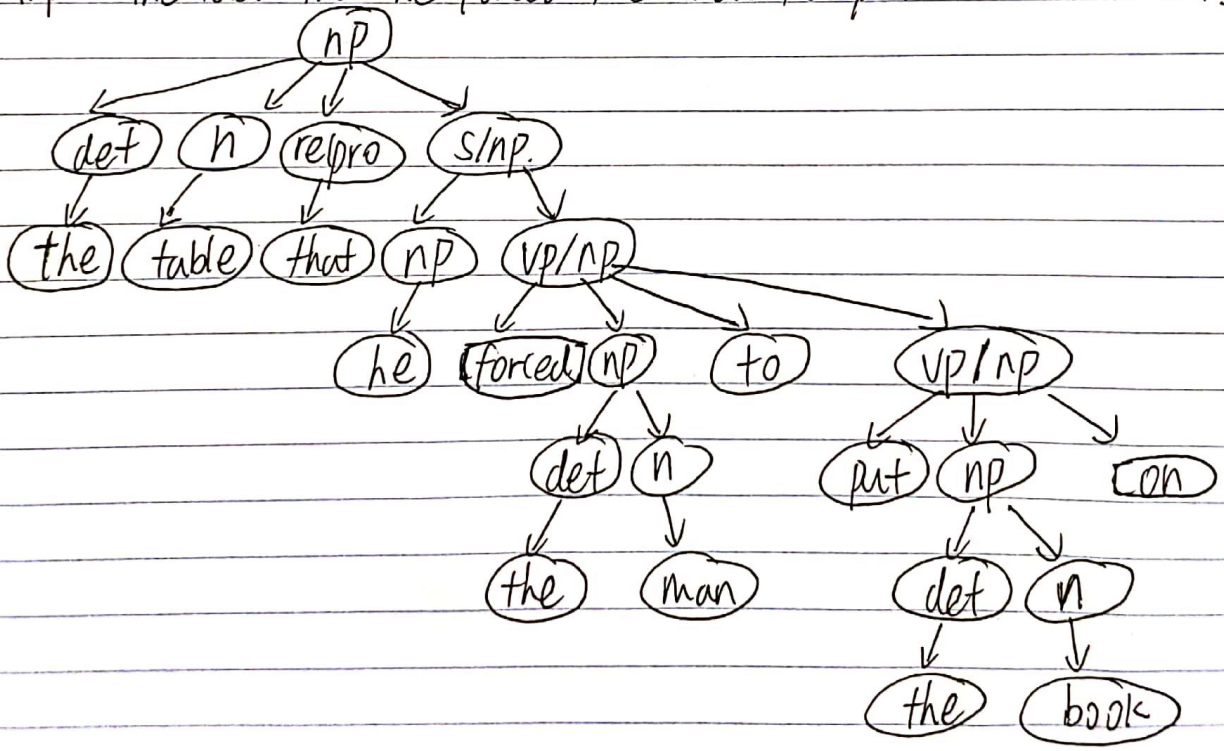


see the rest on next pages

input: the model that the book described



input: the table that he forced the man to put the book on.



Q2(d)

*: I found and tested that there has a default copy constructor in C++ which allows me to copy an object from another like:

Rule new_r = r;

When I changed the values in new_r, r won't change.

```
vector<Rule> gap_version(Rule r){
  vector<Rule> rules;
  for(int i=0; i<r.dtrs.size(); i++){
      if(r.dtrs[i].cat == "np"){
  // if this dtr is np, then create a new rule which copied from r
      Rule new_r = r;

      // change new_r's mother from "X" to "X/np".
      string str = r.mother.cat + "/np";
      Category new_mother(str);
      new_r.mother = new_mother;

      //swap this np to be the last element in new_r.dtrs vector
      for(int j=i; j<new_r.dtrs.size()-1; j++){
          Category tmp = new_r.dtrs[j];
          new_r.dtrs[j] = new_r.dtr[j+1];
          new_r.dtrs[j+1] = tmp;
      }

      // pop the last element in this vector which is np now
      new_r.dtrs.pop_back();

      // push this modified rule into the rules vector.
      rules.push_back(new_r);
    }
  }
  return rules;
}
```