Manxueying Li : uni : ml4529

problem 1

a) words = "they are baking potatoes"

tags1 = PRP, V, Adj, N

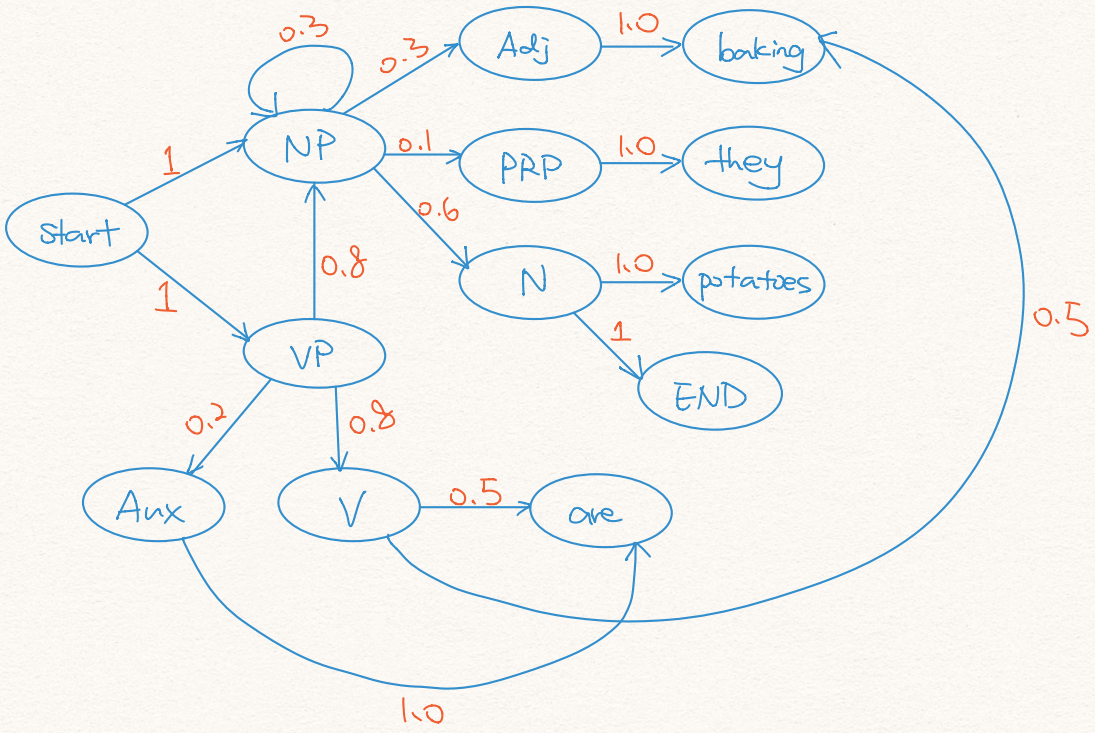
tags2 = PRP, Aux, V, N

Accordig to Q2.b :

P (tags1, words) = 0.0072

P (tags2, words) = 0.006

b)



c) No. PCFGs can express all probabilistic automata, meaning it can represent all CFGs, while HMMs can only represent finate state automata, so that it can only represent regular grammars. And because regular grammar is just a subset of CFGs, therefore not all PCFGs can be translated to HMMs.

problem 2:

(a)

chart[0]

S → .N VP [0,0] – predict

N → .potato [0,0] – nothing

VP → .V NP [0,0] – predict

VP → .Aux V NP [0,0] – predict

V → . baking [0,0] – nothing

V → . are [0,0] – nothing

NP → .Adj NP [0,0] – predict

NP → .PRP [0,0] – predict

NP → .N [0,0] – predict

Aux → .are [0,0] – nothing

Adj → . baking [0,0] – nothing

PRP → .they [0,0] – scan

chart[1]

PRP → they. [0,1] – complete

NP → PRP. [0,1] – complete

VP → .V NP [0,1] – predict

VP → .Aux VNP [0,1] – predict

NP → .Adj NP [0,1] – predict

V → . baking [1,1] – nothing

V → . are [1,1] – scan

Aux → . are [1,1] – scan

Adj → .baking [1,1] – nothing

chart[2]

V → are. [1,2] – complete

Aux → are. [1,2] – complete

VP → V. NP [1,2] – predict

VP → Aux. V NP [1,2] – predict

NP → .Adj. NP [2,2] – predict

NP → .PRP [2,2] – predict

NP → .N [2,2] – predict

Adj → .baking [2,2] – scan

PRP → .they [2,2] – nothing

N → .potatoes [2,2] – nothing

chart[3]

Adj → baking. [2,3] – complete

NP → Adj. NP [2,3] – predict

Adj → .baking [3,3] – nothing

NP → .PRP [3,3] – predict

NP → .N [3,3] – predict

PRP → .they [3,3] – nothing

N → .potatoes [3,3] – scan

chart[4]

N → potatoes. [3,4] – complete

NP → N. [3,4] – complete

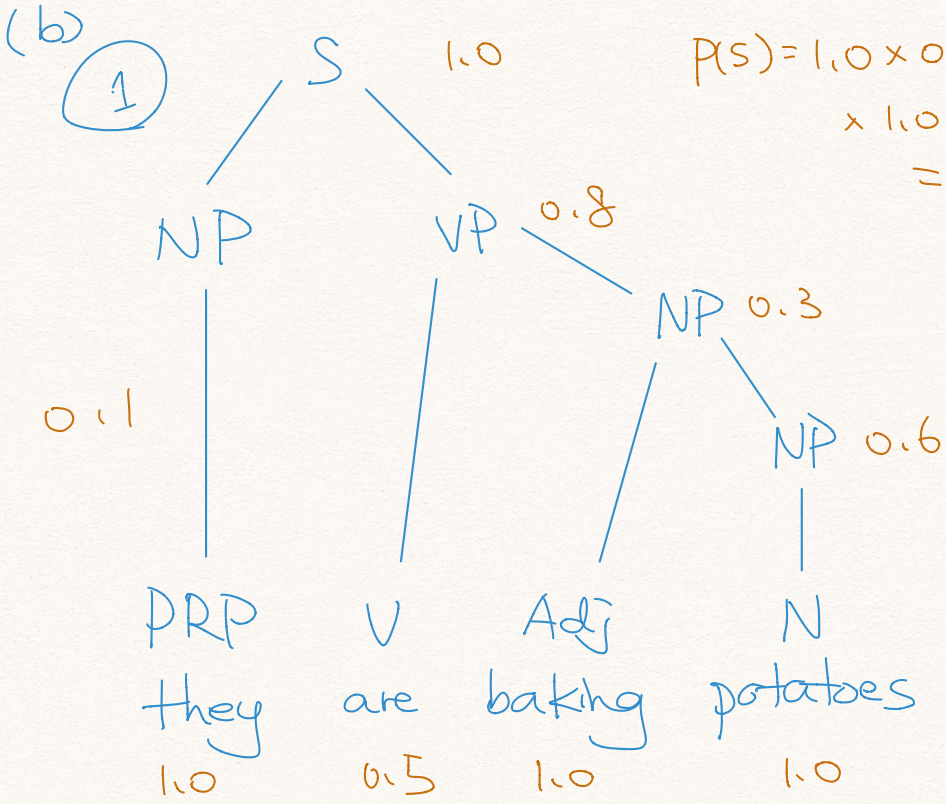
NP → Adj NP. [2,4] – complete

VP → V NP. [1,4] – complete

VP → Aux V NP. [1,4] – complete

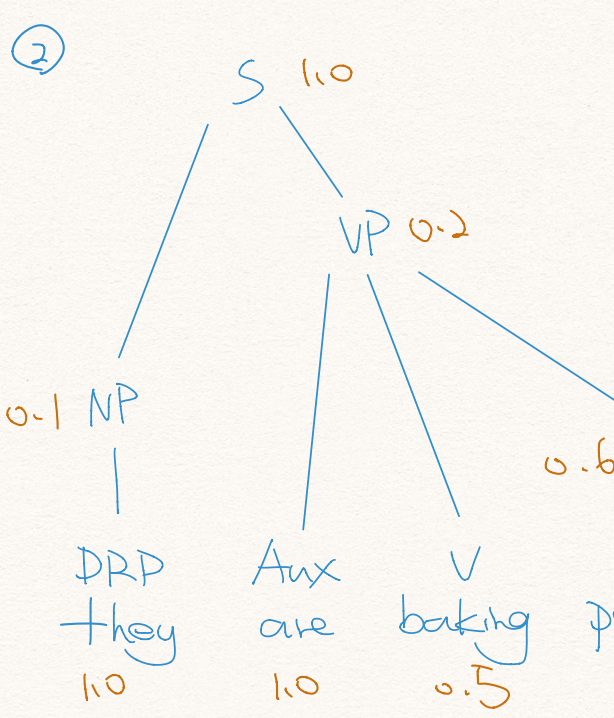
S → N VP. [0,4]

(b)

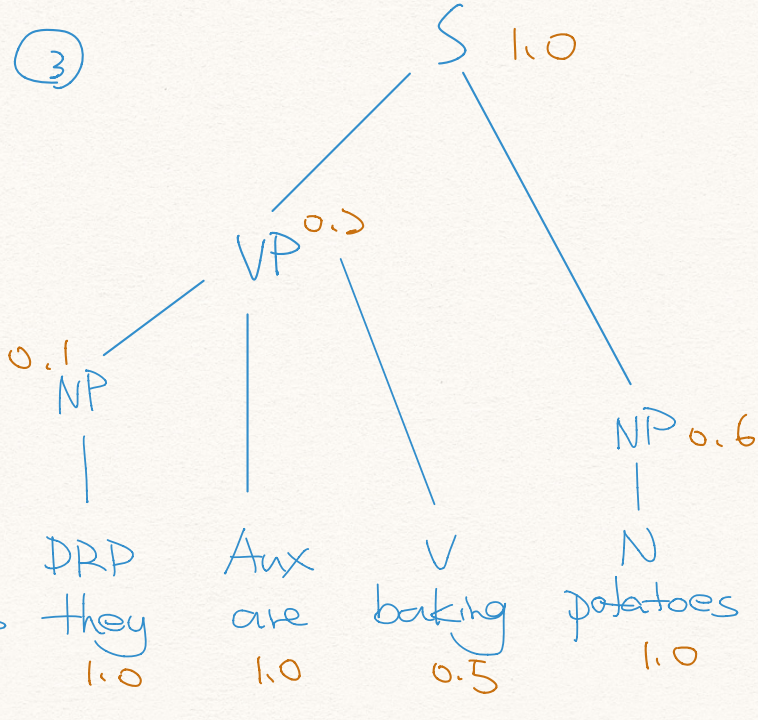


$P(S) = 1.0 \times 0.1 \times 0.8 \times 0.3 \times 0.6 \times 1.0 \times 0.5 \times 1.0 \times 1.0 = 0.0072$

S → NP VP	[1.0]
NP → Adj NP	[0.3]
NP → PRP	[0.1]
NP → N	[0.6]
VP → V NP	[0.8]
VP → Aux V NP	[0.2]
PRP → they	[1.0]
N → potatoes	[1.0]
Adj → baking	[1.0]
V → baking	[0.5]
V → are	[0.5]
Aux → are	[1.0]



$P(S) = 1.0 \times 0.1 \times 0.2 \times 0.6 \times 1.0 \times 1.0 \times 0.5 \times 1.0 = 0.006$



$P(S) = 1.0 \times 0.1 \times 0.2 \times 0.6 \times 1.0 \times 1.0 \times 0.5 \times 1.0 = 0.006$

Problem 3

S → NP VP	[1.0]
NP → Adj NP	[0.3]
NP → PRP	[0.1]
NP → N	[0.6]
VP → V NP	[0.8]
VP → Aux V NP	[0.2]
PRP → they	[1.0]
N → potatoes	[1.0]
Adj → baking	[1.0]
V → baking	[0.5]
V → are	[0.5]
Aux → are	[1.0]

(a) $S \rightarrow NP\ VP$ ✓

$NP \rightarrow Adj\ NP$ ✓

$NP \rightarrow PRP$, $PRP \rightarrow they$ $\Rightarrow$ $NP \rightarrow they$

$NP \rightarrow N$, $N \rightarrow potatoes$ $\Rightarrow$ $NP \rightarrow potatoes$

$VP \rightarrow Aux\ V\ NP$: $\Rightarrow$ $X \rightarrow Aux\ V$, $VP \rightarrow X\ NP$

$Adj \rightarrow baking$ ✓

$V \rightarrow baking$ ✓

$V \rightarrow are$ ✓

$Aux \rightarrow are$ ✓

general rule:

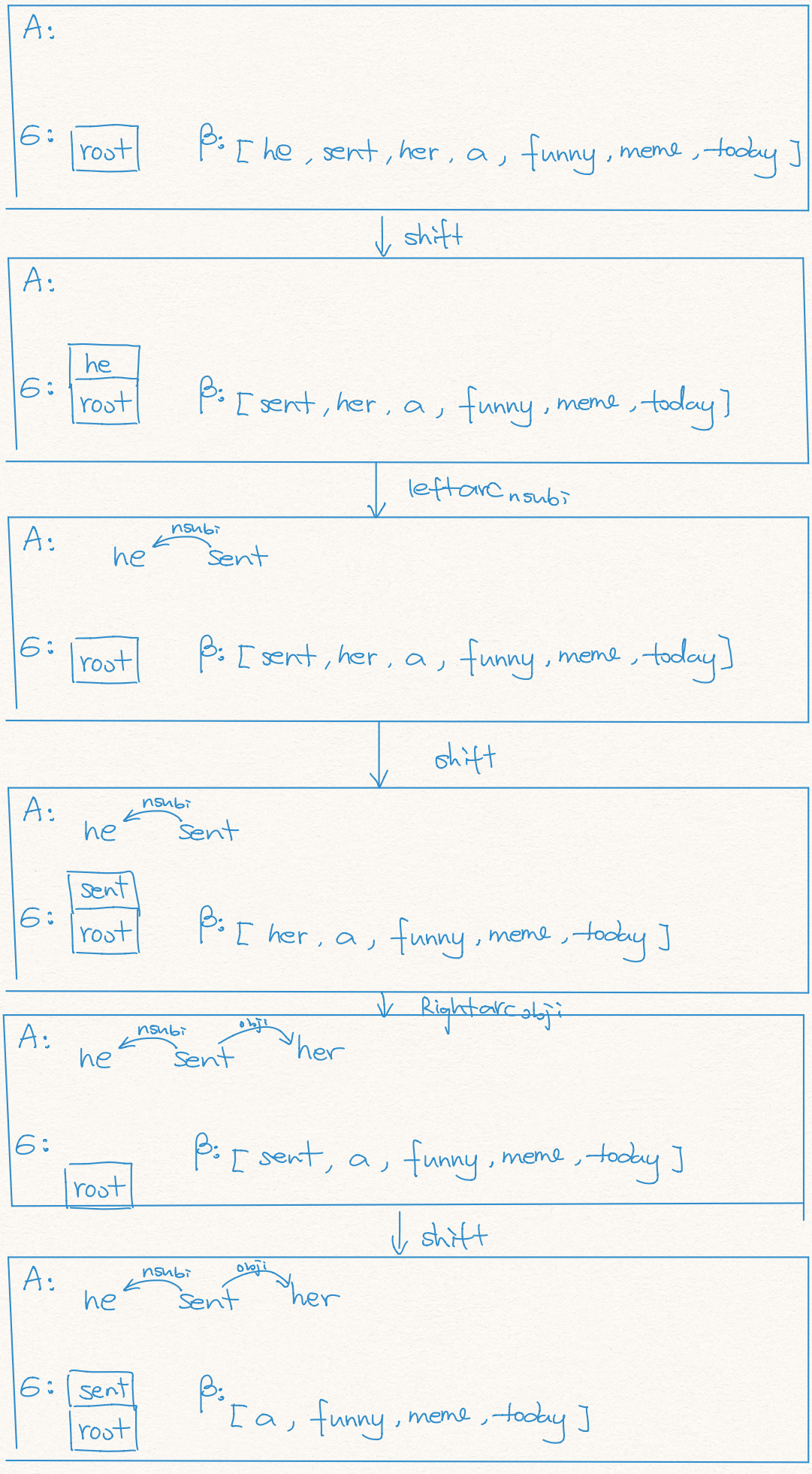
① $A \rightarrow B$, $B \rightarrow b$ } merge to 1: $A \rightarrow b$

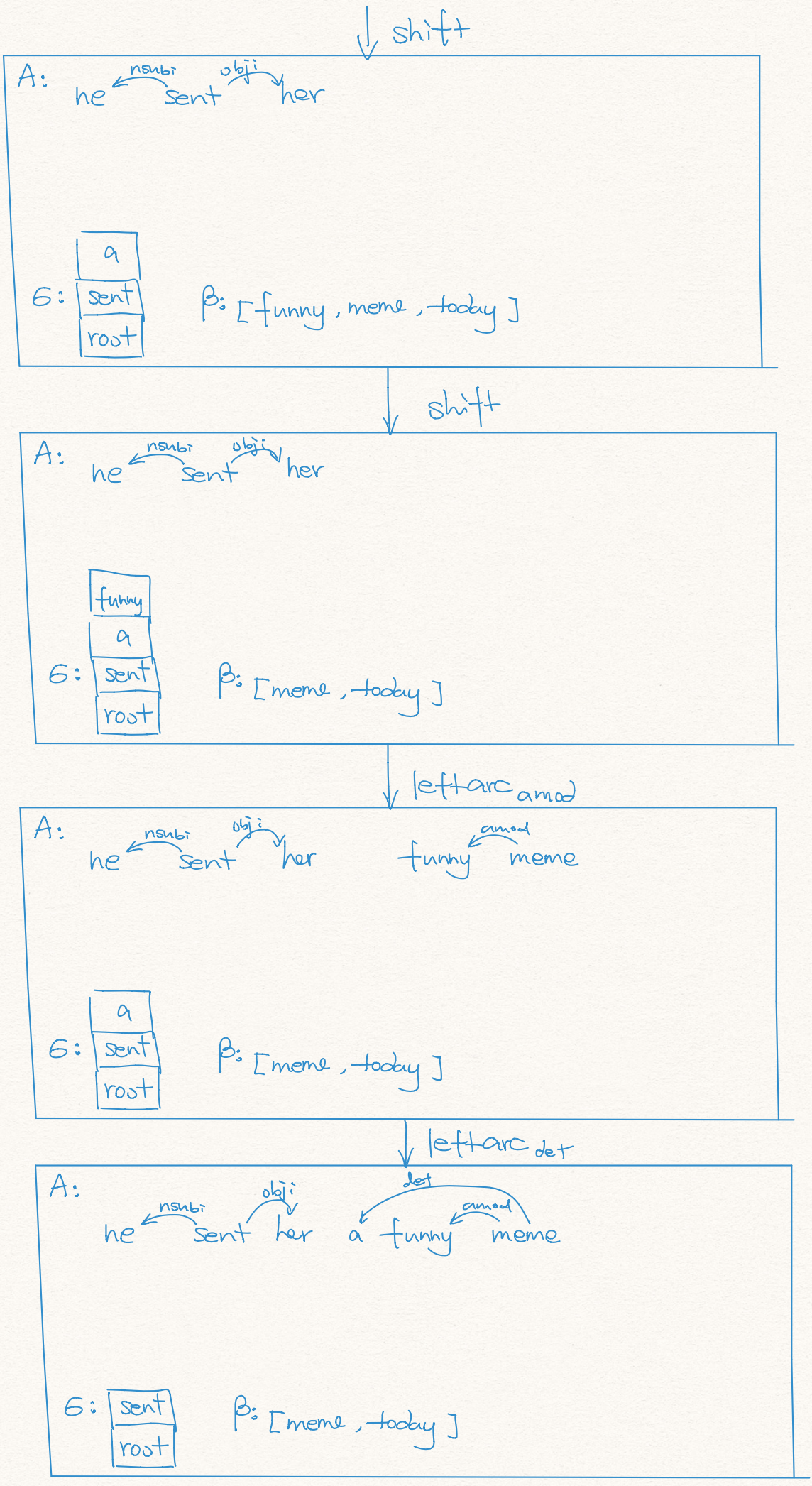
② $A \rightarrow BCD$: { create new rule: $X \rightarrow BC$; use new rule: $A \rightarrow XD$

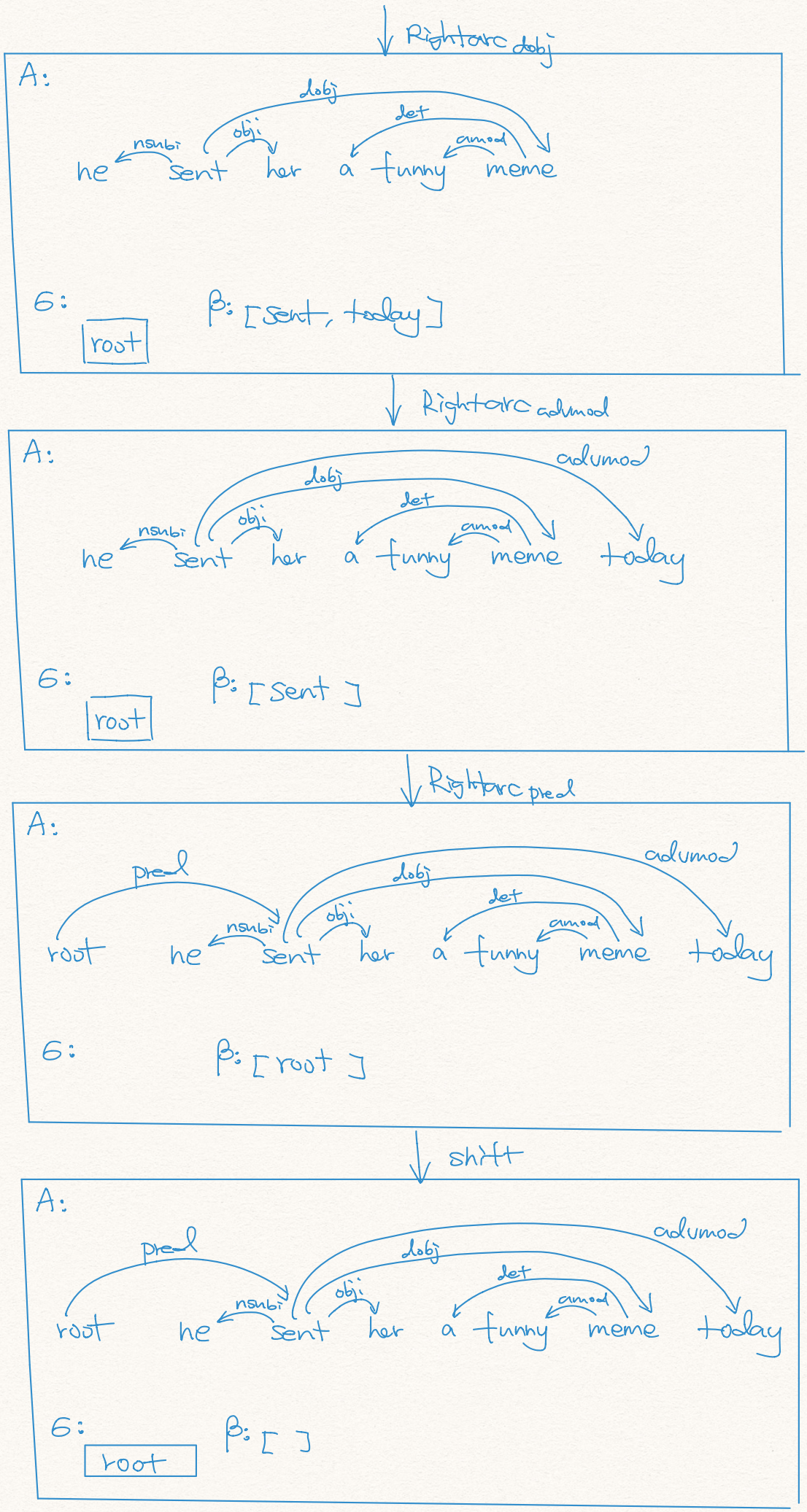
(b)

	0	1	2	3	4
		They	**are**	**baking**	**patatoes**
0	/	NP	/	/	S
1	/	/	Aux, V	X	VP
2	/	/	/	Adj, V	NP
3	/	/	/	/	NP
4	/	/	/	/	/

problem 4


A:
S: root
B: [he , sent , her , a , funny , meme , today]
shift
A:
S: he / root
B: [sent , her , a , funny , meme , today]
leftarc nsubj
A: he ←nsubj– sent
S: root
B: [sent , her , a , funny , meme , today]
shift
A: he ←nsubj– sent
S: sent / root
B: [her , a , funny , meme , today]
Rightarc obj
A: he ←nsubj– sent –obj→ her
S: root
B: [sent , a , funny , meme , today]
shift
A: he ←nsubj– sent –obj→ her
S: sent / root
B: [a , funny , meme , today]


shift
A:
he
nsubj
sent
obj
her
σ:
a
sent
root
β: [funny, meme, today]
shift
A:
he
nsubj
sent
obj
her
σ:
funny
a
sent
root
β: [meme, today]
leftarc amod
A:
he
nsubj
sent
obj
her
funny
amod
meme
σ:
a
sent
root
β: [meme, today]
leftarc det
A:
he
nsubj
sent
obj
her
a
det
funny
amod
meme
σ:
sent
root
β: [meme, today]


Rightarc dobj
A:
dobj
det
nsubj
obj
amod
he sent her a funny meme
σ:
root
β: [sent, today]
Rightarc advmod
A:
advmod
dobj
det
nsubj
obj
amod
he sent her a funny meme today
σ:
root
β: [sent]
Rightarc pred
A:
pred
advmod
dobj
det
nsubj
obj
amod
root he sent her a funny meme today
σ:
β: [root]
shift
A:
pred
advmod
dobj
det
nsubj
obj
amod
root he sent her a funny meme today
σ:
root
β: [ ]

Programming Component:

part1

```
def verify_grammar(self):
        """
        Return True if the grammar is a valid PCFG in CNF.
        Otherwise return False.
        """
        # TODO, Part 1
        for lhs, rule in self.lhs_to_rules.items():
            #lhs should always be upper
            if not lhs.isupper():
                return False
            prob_lst = []
            for curr in rule:
                #total length should be 3 (lhs, rhs, prob)
                #grammar in tuple
                #prob should be float
                #length 2 on rhs + all upper
                #length 1 on rhs + lower
                if (len(curr) != 3)\
                 or not(isinstance(curr[1], tuple))\
                 or not (isinstance(curr[2], (int, float)))\
                 or not ((len(curr[1]) == 2 and curr[1][0].isupper() and curr[1]
[1].isupper()))\
                 or not (len(curr[1]) == 1 and not curr[1][0].isupper()):
                 return False
                else:
                 prob_lst.append(curr[2])

            #prob sum to 1
            if not isclose(fsum(prob_lst), 1.):
                return False

        return True
```

part2

```
def is_in_language(self, tokens):
        """
        Membership checking. Parse the input tokens and return True if
        the sentence is in the language described by the grammar. Otherwise
        return False
        """
        # TODO, part 2
        n = len(tokens)
        pi = defaultdict(list)
        #initialization
        for i in range(n):
            for curr in self.grammar.rhs_to_rules[(tokens[i],)]:
                if len(curr[1]) == 1:
                    pi[(i, i+1)].append(curr[0])
        #main loop
        for length in range(2, n + 1):
            for i in range(n-length+1):
                j = i + length
                for k in range(i+1, j):
                    for B in pi[(i, k)]:
                        for C in pi[(k, j)]:
                            rules = self.grammar.rhs_to_rules[(B, C)]
                            for curr in rules:
                                pi[(i, j)].append(curr[0])

        return self.grammar.startsymbol in pi[(0, n)]
```

part3

```
def parse_with_backpointers(self, tokens):
        """
        Parse the input tokens and return a parse table and a probability table.
        """
        # TODO, part 3
        n = len(tokens)
        table = defaultdict(dict)
        probs = defaultdict(dict)
        #initialization
        for i in range(n):
            for curr in self.grammar.rhs_to_rules[(tokens[i],)]:
                if len(curr[1]) == 1:
                    span = (i, i+1)
                    table[span][curr[0]] = tokens[i]
                    probs[span][curr[0]] = math.log(curr[2])
        #main loop
        for length in range(2, n + 1):
            for i in range(n-length+1):
                j = i + length
                for k in range(i+1, j):
                    for B in table[(i, k)]:
                        for C in table[(k, j)]:
                            rules = self.grammar.rhs_to_rules[(B, C)]
                            if rules:
                                B_prob = probs[(i, k)][B]
                                C_prob = probs[(k, j)][C]
                                for curr in rules:
                                    curr_prob = math.fsum([math.log(curr[2]), B_prob,
C_prob])
                                    span = (i, j)
                                    if curr[0] not in probs[span] or curr_prob >
probs[span][curr[0]]:
                                        table[span][curr[0]] = ((B,i,k),(C,k,j))
                                        probs[span][curr[0]] = curr_prob
        return table, probs
```

part4

```
def get_tree(chart, i, j, nt):
    """
    Return the parse-tree rooted in non-terminal nt and covering span i,j.
    """
    # TODO: Part 4
    # if i >= j or nt not in chart[(i, j)]:
    #     return tuple([])
    if j == i + 1:
        return (nt, chart[(i, j)][nt])

    left_nt, left_i, left_j = chart[(i, j)][nt][0]
    right_nt, right_i, right_j = chart[(i, j)][nt][1]
    return (nt, get_tree(chart, left_i, left_j, left_nt), get_tree(chart, right_i,
right_j, right_nt))
```