Analytical Component:

Problem 1:

a) P(tags, words) = PCFG of Parse Tree #1 + PCFG of Parse Tree #2 (solved in problem 2b)
= 0.006 + 0.0072
= 0.0132

b)

PCFG from problem

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]

Transition Probabilities:

P(Adj|Start) = P(S → NP VP)*P(NP → Adj NP) = 1.0*0.3 = 0.3
P(PRP|Start) = P(S → NP VP)*P(NP → PRP) = 1.0*0.1 = 0.1
P(N|Start) = P(S →NP VP)*P(NP → N) = 1.0*0.6 = 0.6

P(PRP|Adj) = P(NP → Adj NP → Adj PRP) = 0.1
P(N|Adj) = P(NP → Adj NP → Adj N) = 0.6
P(Adj|Adj) = P(NP → Adj NP → Adj Adj NP) = 0.3

P(V|PRP) = P(VP → V NP → V PRP) = 0.8
P(Aux|PRP) = P(NP VP → PRP VP → PRP Aux V NP) = 1.0*0.2 = 0.2

P(V|N) = P(NP VP → N VP → N V NP) = 1.0*0.8 = 0.8
P(Aux|N) = P(NP VP → N VP → N Aux V NP) = 1.0*0.2 = 0.2

P(V|Aux) = P(Aux → V → are) = 1.0

P(PRP|V) = P(VP → V NP → V PRP) = 0.1
P(N|V) = P(VP → V NP → V N) = 0.6
P(Adj|V) = P(VP → V NP → V Adj NP) = 0.3

Emission Probabilities:

P(they|PRP) = P(PRP → they) = 1.0
P(potatoes|N) = P(N → potatoes) = 1.0
P(baking|Adj) = P(Adj → baking) = 1.0
P(baking|V) =P(V → baking) = 0.5
P(are|V) = P(V → are) = 0.5
P(are|Aux) = P(Aux → are) = 1.0

c) No, it is not possible to translate any PCFG into an HMM that produces the identical joint probability as the PCFG. PCFGs generates languages that can be expressed by context free grammar whereas HMMs generates languages that can be expressed by regular grammar. According to the Chomsky Hierarchy, the context free grammar is a generalization of the regular grammar. As such, a grammatical construct that can be described by regular grammar can be described by a context-free grammar, but that is not the case for the other way around. Hence, there are some PCFGs that cannot be translated to an HMM producing identical joint probabilities.

Problem 2:

a)

$_0$ they $_1$ are $_2$ baking $_3$ potatoes $_4$

Chart 0:

S_0 S → • NP VP [0,0] init
S_1 NP → • Adj NP [0,0] predict S_0
S_2 NP → • PRP [0,0] predict S_0
S_3 NP → • N [0,0] predict S_0
S_4 Adj → • baking [0,0] predict S_1
S_5 PRP → • they [0,0] predict S_2
S_6 N → • potatoes [0,0] predict S_3

Chart 1:

S_7 PRP → they • [0,1] scan S_5
S_8 NP → PRP • [0,1] complete S_2 with S_7
S_9 S → NP • VP [0,1] complete S_0 with S_7
S_10 VP → • V NP [1,1] predict S_9
S_11 VP → • Aux V NP [1,1] predict S_9
S_12 V → • baking [1,1] predict S_10
S_13 V → • are [1,1] predict S_10
S_14 Aux → • are [1,1] predict S_11

Chart 2:

S_15 V → are • [1,2] scan S_13
S_16 Aux → are • [1,2] scan S_14
S_17 VP → V • NP [1,2] complete S_10 with S_15
S_18 VP → Aux • V NP [1,2] complete S_11 with S_16
S_19 NP → • Adj NP [2,2] predict S_17
S_20 NP → • PRP [2,2] predict S_17
S_21 NP → • N [2,2] predict S_17
S_22 V → • baking [2,2] predict S_18
S_23 V → • are [2,2] predict S_18
S_24 Adj → • baking [2,2] predict S_19
S_25 PRP → • they [2,2] predict S_20
S_26 N → • potatoes [2,2] predict S_21

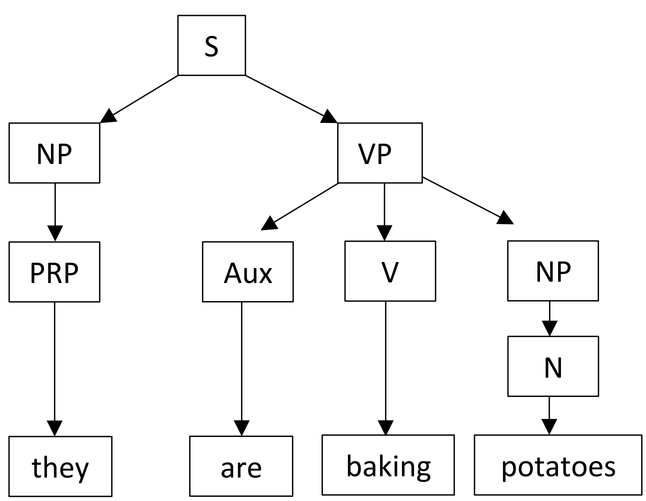
Chart 3:

S_27 V → baking • [2,3] scan S_22
S_28 Adj → baking • [2,3] scan S_24
S_29 VP → Aux V • NP [1,3] complete S_18 with S_27
S_30 NP → Adj • NP [2,3] complete S_19 with S_28
S_31 NP → • Adj NP [3,3] predict S_29, predict S_30
S_32 NP → • PRP [3,3] predict S_29, predict S_30
S_33 NP → • N [3,3] predict S_29, predict S_30
S_34 Adj → • baking [3,3] predict S_31
S_35 PRP → • they [3,3] predict S_32
S_36 N → • potatoes [3,3] precit S_33

Chart 4:

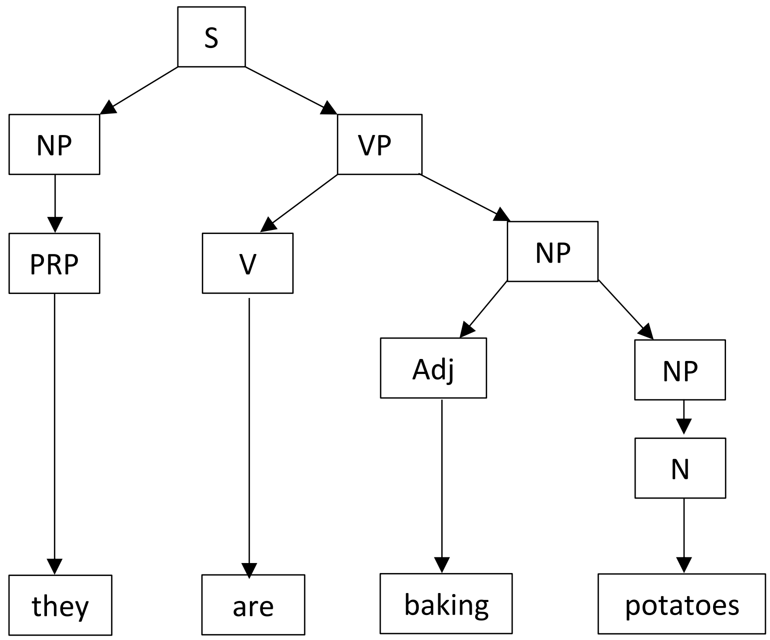
S_37 N → potatoes • [3,4] scan S_36
S_38 NP → N • [3,4] complete S_33 with S_37
S_39 VP → Aux V NP • [1,4] complete S_29 with S_38
S_40 NP → Adj NP • [2,4] complete S_30 with S_38
S_41 S → NP VP • [0,4] complete S_9 with S_39
S_42 VP → V NP • [1,4] complete S_17 with S_40
S_43 S → NP VP • [0,4] complete S_9 with S_42

b)
Final Parsing from part a:
Parse Tree #1



PCFG = 1.0 * 0.1 * 1.0 * 0.2 * 1.0 * 0.5 * 0.6 * 1.0 = 0.006

Parse Tree #2:



PCFG = 1.0 * 0.1 * 1.0 * 0.8 * 0.5 * 0.3 * 1.0 * 0.6 * 1.0 = 0.0072

Problem 3:

a)

- **Rules of the form A→B:** For production rules that have a single non-terminal on the right-hand side **(B)** , the general rule is to replace that non-terminal with a terminal. For instance, say we also have **B → C** where **C** is a terminal. Hence, in **A → B** we replace **B** with **C** and and get rid of the **B → C** rule so that we get to have **A → C.**

- **Rules of the form A→B C D E**: For production rules that have three or more non-terminals on the right-hand side **(B C D E),** the general rule is to separate the non-terminals into sections, each consisting of exactly two non-terminals. Then create new production rules such that there's a single non-terminal on the left-side and maximum of two non-terminals on the right-hand side and use them to change the original rule. For instance, we can set the following rules **L → BC** and **R → DE** such that **L** and **R** are non terminals. Then, we can have the non-terminal **A** in this new rule: **A → LR.**

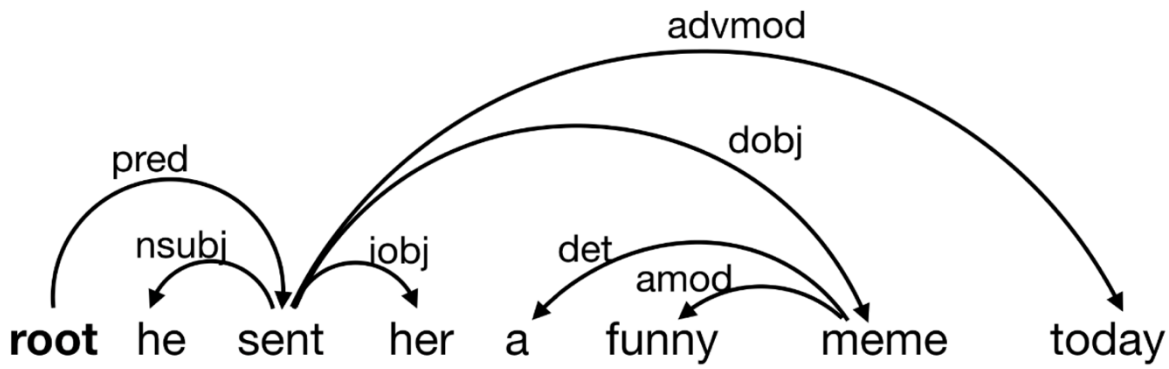
CFG in CNF form:

S → NP VP
NP → Adj NP
NP → they
NP → potatoes
VP → V NP
L → Aux V
VP → L NP
Adj → baking
V → baking
V → are
Aux → are

b) CKY Parse Chart:

	$_0$ they $_1$	are $_2$	baking $_3$	potatoes $_4$
0	NP	VP	VP	S, S, S
1		V, Aux	L	VP
2			Adj, V	NP, VP
3				NP

Problem 4)



1) **Initial State**
   Next Transition: Shift

Stack(σ):

root

Buffer (**β**):

he	sent	her	a	funny	meme	today

A: {}

2) **Next Transition: Left-Arc**

Stack(σ):

he
root

Buffer (**β**):

sent	her	a	funny	meme	today

A: {}

3) Next Transition: Shift

he sent

Stack(σ):

root

Buffer (**β**):

sent	her	a	funny	meme	today

A:{sent, nsubj, he}

4) Next Transition: Right-Arc

he sent

Stack(σ):

sent
root

Buffer (**β**):

her	a	funny	meme	today

A:{sent, nsubj, he}

5) Next Transition: Shift

he sent her

Stack(σ):

root

Buffer (**β**):

sent	a	funny	meme	today

A:{sent, nsubj, he}, {sent, iobj, her}

6) Next Transition: Shift

he sent her

Stack(σ):

sent
root

Buffer (**β**):

a	funny	meme	today

A:{sent, nsubj, he}, {sent, iobj, her}

7) Next Transition: Shift

he sent her

Stack(σ):

a
sent
root

Buffer (**β**):

funny	meme	today

A:{sent, nsubj, he}, {sent, iobj, her}

8) Next Transition: Left-Arc

he sent her

Stack(σ):

funny
a
sent
root

Buffer (**β**):

meme	today

A:{sent, nsubj, he}, {sent, iobj, her}

9) Next Transition: Left-Arc

Stack(σ):

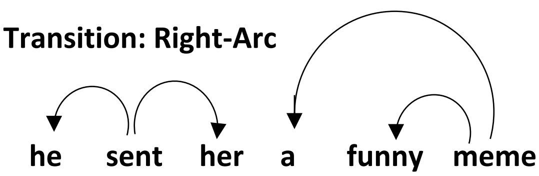
a
sent
root

Buffer (**β**):

meme	today

A:{sent, nsubj, he}, {sent, iobj, her}{meme, amod, funny)

10) Next Transition: Right-Arc



Stack(σ):

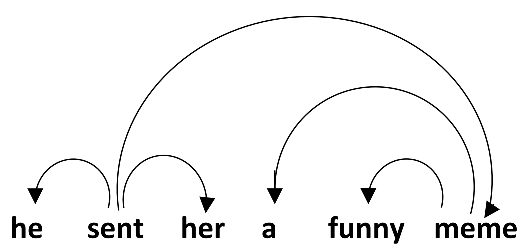
sent
root

Buffer (**β**):

meme	today

A:{sent, nsubj, he}, {sent, iobj, her}{meme, amod, funny}, {meme, det, a}

11) Next Transition: Shift



Stack(σ):

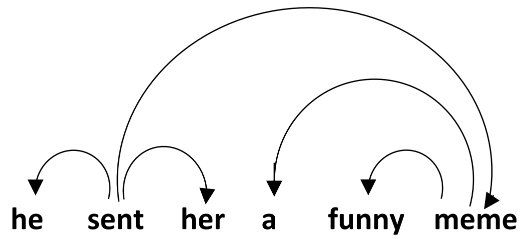
root

Buffer (**β**):

sent	today

A:{sent, nsubj, he}, {sent, iobj, her},{meme, amod, funny},{meme, det, a}, {sent,dobj, meme}

12) Next Transition: Right-Arc

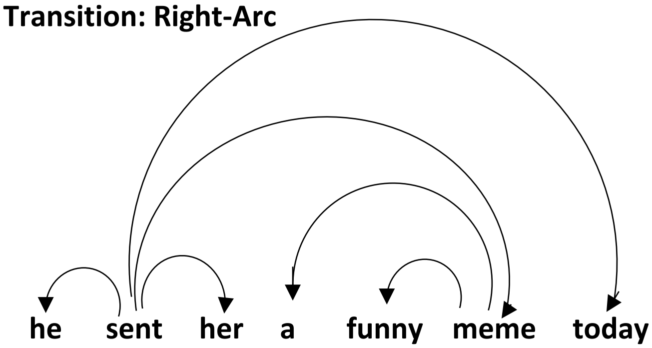


Stack(σ): | sent | root |

Buffer (**β**): | today |

A: :{sent, nsubj, he}, {sent, iobj, her},{meme, amod, funny},{meme, det, a}, {sent,dobj, meme}

13) Next Transition: Right-Arc

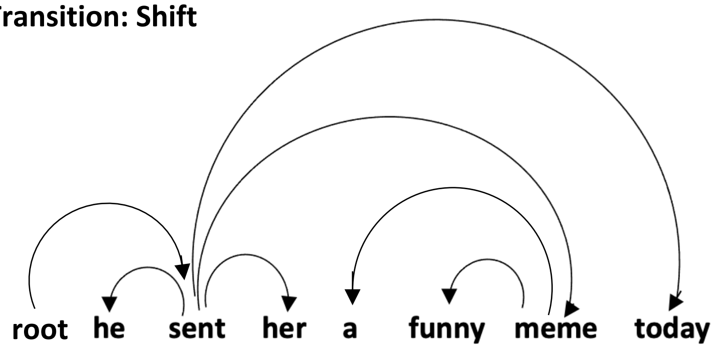


Stack(σ): | root |

Buffer (**β**): | sent |

A: {sent, nsubj, he}, {sent, iobj, her},{meme, amod, funny},},{meme, det, a}, {sent,dobj, meme}, {sent, advmod, today}

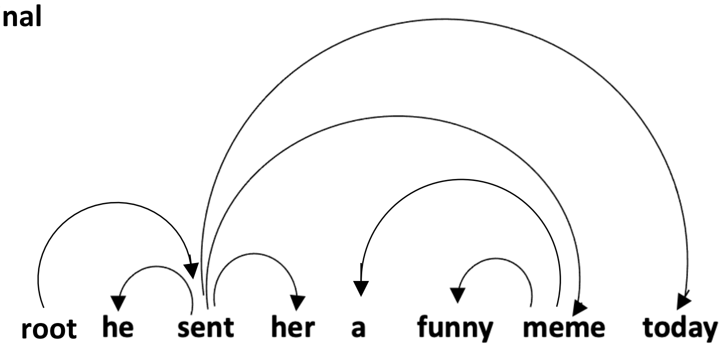
14) Next Transition: Shift



Stack(σ): [ ] Buffer (**β**): [root]

A:{sent, nsubj, he}, {sent, iobj, her},{meme, amod, funny},},{meme, det, a}, {sent,dobj,meme},{sent, advmod, today}, {root, pred, sent}

15) Terminal



Stack(σ): [root] Buffer (**β**): [ ]

A:{sent, nsubj, he}, {sent, iobj, her},{meme, amod, funny}, },{meme, det, a}, {sent,dobj,meme},{sent, advmod, today}, {root, pred, sent}

Programming Component:

Part 1 - reading the grammar and getting started

```
def verify_grammar(self):
    """
    Return True if the grammar is a valid PCFG in CNF.
    Otherwise return False.
    """
    # TODO, Part 1

    lhs_sum = []
    lhs = self.lhs_to_rules #lists of rules(tuples) where tuple[0] is the lhs
    #key : non terminal, v : list of its tuples

    for non_terminal, list in lhs.items():
        for tuple in list:
            #check if non-terminal is uppercase
            if tuple[0].isupper():
                prob = tuple[2]
                lhs_sum.append(prob)
        #check if sum == 1
        print(round(math.fsum(lhs_sum), 2))
        if round(math.fsum(lhs_sum), 2) == 1.0:
            lhs_sum = []
        else:
            return False

    return True
```

Part 2 - Membership checking with CKY

```
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) #number of words
    table = defaultdict() #initialization

    for i in range(0, n):
        for j in range(i + 1, n + 1):
            table[(i, j)] = defaultdict() #initialization

            if i + 1 == j:  # diagonal tuple
                if self.grammar.rhs_to_rules.get((tokens[i],)):
                    k = (tokens[i],)
                    rules = self.grammar.rhs_to_rules[k]
                    #print("k = ", k)
                    #print("rules = ", rules)
                    for rule in rules:
                        table[(i, j)][rule[0]] = tokens[i]

    for length in range(2, n + 1):
        for i in range(0, n - length + 1):
            j = i + length
            for k in range(i + 1, j):
                for key in self.grammar.rhs_to_rules.keys():
                    for t1 in table[(i, k)]:
                        for t2 in table[(k, j)]:
                            if key == (t1, t2):
                                #print("key = ", key)
                                rules =
self.grammar.rhs_to_rules.get(key)
                                #print("rules = ", rules)
                                for r in rules:
                                    #print((t1, i, k), (t2, k,
j))
                                    table[(i, j)][r[0]] = ((t1,
i, k), (t2, k, j))

    if self.grammar.startsymbol in table[(0,n)]:
        return True
```

```
    else:
        return False
```

Part 3 - Parsing with backpointers

```
def parse_with_backpointers(self, tokens):
    """
    Parse the input tokens and return a parse table and a
probability table.
    """
    # TODO, part 3

    """
    TODO: Write the method parse_with_backpointers(self,
tokens). You should modify your CKY implementation from part 2,
but use
    (and return) specific data structures. The method should
take a list of tokens as input and
    returns a) the parse table b) a probability table. Both
objects should be constructed during
    parsing. They replace whatever table data structure you used
in part 2.
    """

    table= defaultdict()  #initialization
    probs = defaultdict() #initialization
    n = len(tokens)   # number of words

    for i in range(0, n):
        for j in range(i+1, n+1):
            table[(i, j)] = defaultdict() #initialization
            probs[(i, j)] = defaultdict() #initialization

            if i + 1 == j: #diagonal tuple
                if self.grammar.rhs_to_rules.get((tokens[i],)):
                    k = (tokens[i],)
                    rules = self.grammar.rhs_to_rules[k]
                    for rule in rules:
                        table[(i, j)][rule[0]] = tokens[i]
                        probs[(i, j)][rule[0]] =
math.log(rule[2])

    for length in range(2, n + 1):
        for i in range(0, n - length + 1):
            j = i + length
            for k in range(i + 1, j):
                for key in self.grammar.rhs_to_rules.keys():
                    for t1 in table[(i, k)]:
                        for t2 in table[(k, j)]:
                            if key == (t1,t2):
```

```
                                rules =
self.grammar.rhs_to_rules.get(key)
                                p = probs[(i, k)][t1] +
probs[(k, j)][t2]

                                for r in rules:
                                    log_prob = math.log(r[2]) +
p
                                    if r[0] in table[(i, j)]:
                                        #check for the highest
probability value
                                        if log_prob > probs[(i,
j)][r[0]]:
                                            probs[(i, j)][r[0]]
= log_prob
                                            table[(i, j)][r[0]]
= ((t1, i, k), (t2, k, j))
                                    else:
                                        probs[(i, j)][r[0]] =
log_prob
                                        table[(i, j)][r[0]] =
((t1, i, k), (t2, k, j))
    return table, probs
```

Part 4 - Retrieving a parse tree

```
def get_tree(chart, i,j,nt):
    """
    Return the parse-tree rooted in non-terminal nt and covering 
span i,j.
    """
    # TODO: Part 4

    backpointers = chart[(i,j)][nt]
    #print(backpointers)
    if type(backpointers) != str:
        result = (nt,
                  (get_tree(chart, i=backpointers[0][1], 
j=backpointers[0][2], nt=backpointers[0][0])),
                  (get_tree(chart, i=backpointers[1][1], 
j=backpointers[1][2], nt=backpointers[1][0])))
        return result
    else:
        result = (nt, backpointers)
        return result
```

Part 5 - Evaluating the Parser

Since we were not required to write code in evaluate_parser.py and to only run it, this is what my parser implementation produces on the atis3 test corpus:

```
Coverage: 67.24%, Average F-score (parsed sentences):
0.9526771952649747, Average F-score (all sentences):
0.6405932864712761
```