Mid-Semester Test Solutions

Each answer was marked out of 3. A mark of 2 indicated that something was missing from the answer, such as an example, some aspect of a term which was not explained, some other omission, or extraneous information. A mark of 1 indicated that the answer was only vaguely relevant, and only touched on the expected answer. Parts B and C were weighted double that of part A.

Mark breakdown:

Part A: /18	Part B: /18	Part C: /18	Total: /72

Part A: Key Concepts

1. Give short definitions of the following terms as they are used in the context of language and computation, with the help of examples:

   (a) n-gram part-of-speech tagger
   A program which annotates each token of a text with its syntactic category. It uses the $n-1$ preceding tags and the current token, in order to predict the most likely tag for the token in this context. E.g. a bigram tagger could tag the word 'deal' as VB when preceded by the tag TO but as NN when preceded by DT.

   (b) recursive syntactic structure
   Sentences can be built up out of other sentences, e.g. 'I think (she is sleeping)'. Noun phrases can contain noun phrases, e.g. 'the tree on (the cliff)'. This is significant since it enables us to build an infinite variety of sentences using a finite grammar.

   (c) structural ambiguity
   This occurs when a sentence has more than one possible syntactic structure. E.g. the sentence 'I saw the building with a telescope', the prepositional phrase 'with a telescope' could be linked to 'saw' (describing how I did it), or to 'building' (describing which particular building I saw).

2. Explain the following concepts and the distinctions between them, with the help of examples:

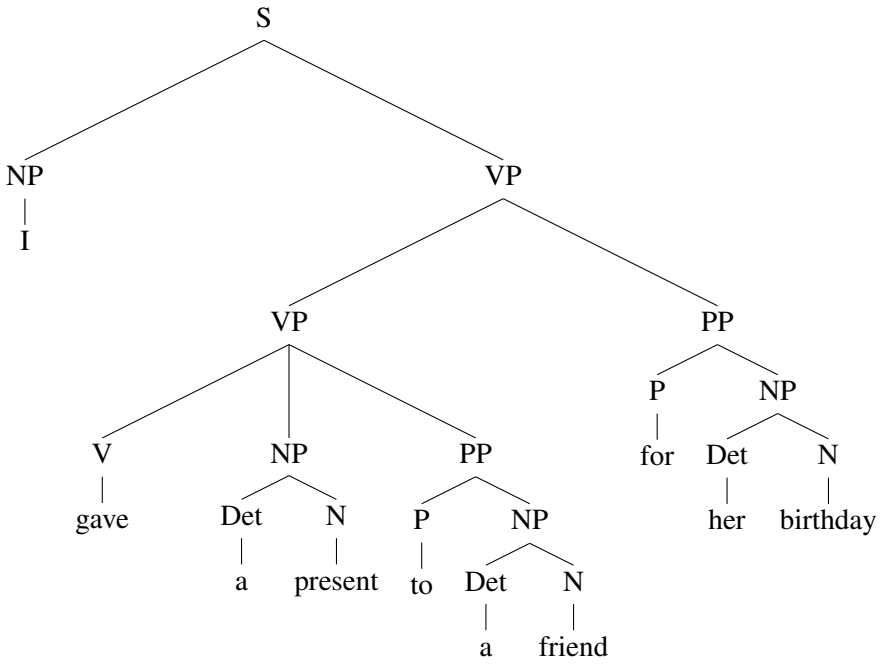
   (a) transitive *vs* intransitive verb
   *These are two kinds of 'valency' or 'subcategorization'. An intransitive verb, like 'sleep', requires a single noun phrase and it is ungrammatical to provide two, e.g. '*I slept it'. A transitive verb, like 'chase', requires two noun phrases, e.g. 'I chased it'.*

   (b) syntactic well-formedness *vs* semantic interpretability
   A sentence is syntactically well-formed if it obeys the rules of grammar. A sentence is semantically interpretable if a listener can work out a meaning for it. We can have one without the other, e.g. 'colourless green ideas sleep furiously' is grammatical nonsense, while 'I can has cheeseburger' is meaningful but ungrammatical.

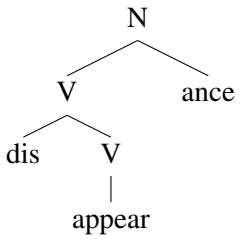
   (c) parser *vs* grammar
   A grammar is a formal definition of syntactic well-formedness, which usually consists of a set of productions like S → NP VP. A parser is a program which analyzes a sentence according to a grammar, and returns the syntactic structure it found, e.g. (S (NP she) (VP (V chased) (NP it))).

Part B: Language Analysis

3. Draw a tree diagram to represent the constituent structure of the sentence: *I gave a present to an old friend for her birthday*



4. What is the morphological structure of the word: *disappearance*



5. Explain the process of training a unigram tagger; show the internal data structure that would be built up while processing the following training data:

```
They/PR refuse/V to/TO permit/V us/PR to/TO obtain/V the/D refuse/N permit/N
```

A unigram tagger guesses the most likely syntactic category for a word (regardless of its context). It is trained using POS-tagged text. It counts the number of times a given word occurs with a given tag:

Word	Observed Tags
They	PR: 1
refuse	V: 1, N: 1
to	TO: 2
permit	V: 1
us	PR: 1
obtain	V: 1
the	D: 1
permit	N: 1

Part C: Python Programming

6. Explain the purpose of the following code:

```
def mystery(s):
    d = {'v':0, 'c':0}
    for w in s:
        if w.isalpha():
            if w in "aeiou":
                key = "v"
            else:
                key = "c"
            d[key] += 1
    return float(d["v"]) / d["c"]
```

The code works out the ratio of vowels to consonants in a given word.

7. Suppose that a text is represented as a list of words and punctuation symbols (i.e. "tokens"), where each token is represented as a string. Assume the text `text1` is already defined. Write code to find all words of `text1` that contain at least two capital letters (e.g. *CNet, R.E.M., R&D, DayGlo*).

```
[w for w in text1 if re.match(r'[A-Z].*[A-Z]', w)]
```

OR:

```
for word in text1:
    if re.match(r'[A-Z].*[A-Z]', word):
        print word
```

OR:

```
result = []
for word in text1:
    uppercase = 0
    for char in word:
        if char.isupper():
            uppercase += 1
    if uppercase >= 2:
        result.append(word)
print result
```

8. Define a function `longest(text)` that takes a text (represented as a list of tokens), and returns the length of the longest word in `text`.

```
def longest(text):
    longest = 0
    for word in text:
        if len(word) > longest:
            longest = len(word)
    return longest
```

OR:

```
def longest(text):
    return max(len(w) for w in text)
```