Homework #5: CMPT-413

Reading: NLTK Tutorial (*http://nltk.sf.net/docs.html*); Chapter 8 and Chapter 9

Distributed on Mar 7; due on Mar 21

Anoop Sarkar – anoop@cs.sfu.ca

Only submit answers for questions marked with †.

(1) † Write down a grammar that derives 5 parse trees for the sentence:

Kafka ate the cookie in the box on the table.

Also provide an implementation of the Catalan number function, and use the Catalan function to predict the number of parses reported for the above sentence. You should insert your grammar and the implementation of the catalan function into the following code fragment (the code is a skeleton and will not work without modification):

```
from nltk_lite.parse import cfg, chart

# implement the catalan function and insert the right value for the
# parameter n below so that you can predict the total number of parses
# for the sentence "Kafka ate the cookie in the box on the table."
print "catalan number =", catalan(n)

grammar = cfg.parse_grammar("""
# write down your grammar here in the usual X -> A B C format
""")
tokens = ["Kafka", "ate", "the", "cookie", "in", "the", "box", "on", "the", "table"]
cp = chart.ChartParse(grammar, chart.TD_STRATEGY)
c = 0
for tree in cp.get_parse_list(tokens):
    print tree
    c += 1
# note that the number of trees below should match the output
# of the catalan function above
print "number of parse trees=", c
```

(2) Provide a verbal description (a paraphrase) of a suitable *meaning* for each of the five possible parse trees produced by your grammar in Question (1) for the sentence:

Kafka ate the cookie in the box on the table.

(3) † Pingala, a linguist who lived circa 5th century B.C., wrote a treatise on Sanskrit prosody called the Chandas Shastra. Virahanka extended this work around the 6th century A.D., providing the number of ways of combining short and long syllables to create a meter of length n. Meter is the recurrence of a pattern of short and long syllables (or stressed and unstressed syllables when applied to English). He found, for example, that there are five ways to construct a meter of length 4: V_4 = {LL, SSL, SLS, LSS, SSSS}. In general, we can split V_n into two subsets, those starting with L: {LL, LSS}, and those starting with S: {SSL, SLS, SSSS}. This provides a recursive definition for computing the number of meters of length n. Virahanka wrote down the following recursive definition for a 6th century Python interpreter:

```
def virahanka(n, lookup = {0:[""], 1:["S"]} ):
    if (not lookup.has_key(n)):
        s = ["S" + prosody for prosody in virahanka(n-1)]
        l = ["L" + prosody for prosody in virahanka(n-2)]
        lookup.setdefault(n, s + l) # insert a new computed value as default
    return lookup[n]
```

Note that Virahanka has used top-down dynamic programming to ensure that if a user calls `virahanka(4)` then the two separate invocations of `virahanka(2)` will only be computed once: the first time `virahanka(2)` is computed the value will be stored in the `lookup` table and used for future invocations of `virahanka(2)`.

Write a *non-recursive* bottom-up dynamic programming implementation of the `virahanka` function. Provide the following output:

```
virahanka(4) = ['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

(4) † Implement the Earley recognition algorithm. Note that you only have to accept or reject an input string based on the input grammar. You do not have to produce the parse trees for a valid input string.

Use the following grammar:

```
from nltk_lite import parse

gram = '''
S -> NP VP
VP -> V NP | V NP PP
V -> "saw" | "ate"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "dog" | "cat" | "cookie" | "park"
PP -> P NP
P -> "in" | "on" | "by" | "with"
'''
g = parse.cfg.parse_grammar(gram)
```

Produce a trace of execution for the input sentence using your implementation of the Earley algorithm:

> the cat in the park ate my cookie

The file `earley-setup.py` contains some helpful tips on how to build the data structures required to implement this algorithm. Your program should produce a trace that is identical to the output given in the file `earley-trace.txt`.

(5) † Using the nltk_lite functions for feature structures (see Section 9.2.6 of the NLTK tutorials), provide a Python program that prints out the results of the following unifications:

1. $\left[\textbf{number: sg}\right] \sqcup \left[\textbf{number: sg}\right]$
2. $\left[\textbf{number: sg}\right] \sqcup \left[\textbf{person: 3}\right]$
3. $\begin{bmatrix}\textbf{agreement:} & \boxed{1}\left[\textbf{number: sg}\right] \\ \textbf{subject:} & \left[\textbf{agreement: } \boxed{1}\right]\end{bmatrix} \sqcup \left[\textbf{subject: } \left[\textbf{agreement: } \left[\textbf{person: 3}\right]\right]\right]$

(6) † Consider the following feature-based context-free grammar:

```
% start S
S -> NP[num=?n] VP[num=?n]
NP[num=?n] -> Det[num=?n] N[num=?n]
NP[num=pl] -> N[num=pl]
VP[tense=?t, num=?n] -> TV[tense=?t, num=?n] NP
Det -> 'the'
N[num=sg] -> 'dog'
N[num=pl] -> 'dogs' | 'children'
TV[tense=pres, num=sg] -> 'likes'
TV[tense=pres, num=pl] -> 'like'
TV[tense=past, num=?n] -> 'liked'
```

The notation `?n` represents a variable that is used to denote reentrant feature structures, e.g., in the rule

```
S -> NP[num=?n] VP[num=?n]
```

the `num` feature of the NP has to be the same value as the `num` feature of the VP.
The notation can also be used to pass up a value of a feature, e.g., in the rule

```
VP[tense=?t, num=?n] -> TV[tense=?t, num=?n] NP
```

the `tense` and `num` features are passed up from the transitive verb TV to the VP.
Save this grammar to the file `feat.cfg`, then we can use the following code to parse sentences using the above grammar.

```
from grammarfile import *

def parse_sent(cp, tokens):
    trees = cp.get_parse_list(tokens)
    print "sentence =", " ".join(tokens)
    if (len(trees) > 0):
        for tree in trees: print tree
    else:
        print "NO PARSES FOUND"

g = GrammarFile.read_file('feat.cfg')
print g.earley_grammar()
cp = g.earley_parser()

parse_sent(cp, ['the', 'dog', 'likes', 'children'])
parse_sent(cp, ['the', 'dogs', 'like', 'children'])
parse_sent(cp, ['the', 'dog', 'like', 'children']) # should not get any parses
parse_sent(cp, ['the', 'dogs', 'likes', 'children']) # should not get any parses
parse_sent(cp, ['the', 'dog', 'liked', 'children'])
parse_sent(cp, ['the', 'dogs', 'liked', 'children'])
```

Note that the above code fragment uses the file `grammarfile` which is provided to you in `~anoop/cmpt413/hw5` (the nltk_lite version has some problems). You can read more about feature-based CFGs in Section 9.2 of the NLTK tutorials, *however do not use the code fragments listed there, use the above code instead*.

Write down a feature-based CFG that will appropriately handle the agreement facts in the Spanish noun phrases shown below; *sg* stands for singular, *pl* stands for plural, *masc* stands for masculine gender, *fem* stands for feminine gender. In Spanish, inanimate objects also carry grammatical gender morphology.

1. un cuadro hermos-o.
   a(sg.masc) picture beautiful(sg.masc).
   'a beautiful picture'
2. un-os cuadro-s hermos-os.
   a(pl.masc) picture(pl) beautiful(pl.masc).
   'beautiful pictures'
3. un-a cortina hermos-a.
   a(sg.fem) curtain beautiful(sg.fem).
   'a beautiful curtain'
4. un-as cortina-s hermos-as.
   a(pl.fem) curtain(pl) beautiful(pl.fem).
   'beautiful curtains'

Provide the feature-based CFG as a text file, and the Python code that reads in the file and parses the above noun phrases and prints out the trees. Include at least two ungrammatical noun phrases that violate the agreement facts and as a result cannot be parsed.

(7) A subcategorization frame for a verb represents the required arguments for the verb, e.g. in the sentence *Zakalwe ate haggis* the verb *eat* has a subcat frame NP. Similarly in the sentence *Diziet gave Zakalwe a weapon* the subcat frame for *give* is NP NP. Assume that we wanted to compactly represent subcategorization frames for verbs using the CFG:

$$\begin{aligned} VP &\rightarrow Verb \\ VP &\rightarrow VP\ X \end{aligned}$$

The non-terminal X can be associated with the category of the various arguments for each verb using a feature structure. For example, for a verb which takes an NP arguments, X is associated with the feature structure: $\left[\text{cat: NP}\right]$. Write down a feature-based CFG that associates with the VP non-terminal an attribute called *subcat* which can be used to compactly represent all of the following subcategorization frames:

1. no arguments
2. NP
3. NP NP
4. NP PP
5. S
6. NP S

Note that the CFG rules should not be duplicated with different feature structures for the different subcategorization frames.

(8) The `WordNet` database is accessible online at `http://www.cogsci.princeton.edu/~wn/`. Follow the link to *Use WordNet Online* or go directly to:
`http://www.cogsci.princeton.edu/cgi-bin/webwn`

`WordNet` contains information about word *senses*, for example, the different senses of the word *plant* as a manufacturing plant or a flowering plant. For each sense, `WordNet` also has several class hierarchies based on various relations. Once such relation is that of *hypernyms* also known as *this is a kind of . . .* relation. It is analogous to a object-oriented class hierarchy over the meanings of English nouns and verbs and is useful in providing varying types of word class information.

For example, the word *pentagon* has 3 senses. The sense of *pentagon* as five-sided polygon has the following hypernyms. The word *line* has 29 senses. The sense of *line* as trace of moving point in geometry has the following hypernyms.

Sense3: pentagon
⇒polygon,polygonal-shape
⇒plane-figure,two-dimensional-figure
⇒figure
⇒shape,form
⇒attribute
⇒abstraction

Sense4: line
⇒shape,form
⇒attribute
⇒abstraction

The *lowest common ancestor* for these two senses is the hypernym shape, form. A *hypernym path* goes up the hypernym hierarchy from the first word to a common ancestor and then down to the second word. Note that a hypernym path from a node other than the lowest common ancestor will always be equal to or longer than the hypernym path provided by the lowest common anscestor. For the above example, the hypernym path is *pentagon* ⇒ polygon,polygonal-shape ⇒ plane-figure,two-dimensional-figure ⇒ figure ⇒ shape,form ⇒ *line*.

Find the lowest common ancestor across all senses of each of the following pairs of words: (a) *Canada* and *Vancouver*, and (b) *English* and *Tagalog*.