UNIVERSITÄT WIEN
CSLEARN - EDUCATIONAL TECHNOLOGIES

Natural Language Processing

Exercise Sheet 4

Writing Structured Programs

Exercise 1

Write a program to initialize a two-dimensional array of sets called `word_vowels` and process a list of words, adding each word to `word_vowels[l][v]` where l is the length of the word and v is the number of vowels it contains. Test your program with a 10x10-array and the list `['Alice', 'hat', 'heute', 'ihren', 'freien', 'Tag']`.

Exercise 2

Write a program that prints all words that only appear in the last 10% of a text. Test your code with the file `'shakespeare-macbeth.txt'` from the Gutenberg Corpus.

Exercise 3

Write a program that takes a sentence expressed as a single string, splits it and counts up the words. Get it to print out each word and the word's frequency, one per line, in alphabetical order. Test it with the sentence: `'das ist heute wieder einmal wirklich ein sehr schöner tag das kann ich dir wieder einmal sagen'`.

Exercise 4

Write a function `sort_dist(candidates, target)`. The `candidates` are a list of strings representing WordNet synset names, and `target` a synset name string. The function shall sort the `candidates` for proximity to the `target` synset using `shortest_path_distance()`.

Test your function with `candidates=['minke_whale.n.01','orca.n.01','novel.n.01', 'tortoise.n.01']` and `target='right_whale.n.01'`.

Exercise 5

Write a recursive function `lookup(trie, key)` that looks up a `key` in a `trie`, and returns the value it finds. The function should cover the following cases:

a) it should return a corresponding message if the key is not included in the trie;

b) it should return a message if the key is not unique, i.e. if there are several words for this prefix;

c) if a word is uniquely determined by the key prefix it should be returned as result.

Try your function for the following trie and test cases:

```
def insert(trie, key, value):
    if key:
        first, rest = key[0], key[1:]
        if first not in trie:
            trie[first] = {}
        insert(trie[first], rest, value)
    else:
        trie['value'] = value

trie = {}
insert(trie, 'chat', 'cat')
insert(trie, 'chien', 'dog')
insert(trie, 'chair', 'flesh')
insert(trie, 'chic', 'stylish')
insert(trie, 'cheval','horse')
trie = dict(trie)
pprint.pprint(trie, width=40)

{'c': {'h': {'a': {'i': {'r': {'value': 'flesh'}},
                   't': {'value': 'cat'}},
             'e': {'v': {'a': {'l': {'value': 'horse'}}}},
             'i': {'c': {'value': 'stylish'},
                   'e': {'n': {'value': 'dog'}}}}}}

print(lookup(trie, 'chat'))
print(lookup(trie, 'cha'))
print(lookup(trie, 'souris'))
print(lookup(trie, 'cheval'))
print(lookup(trie, 'che'))
print(lookup(trie, 'chev'))
```

Exercise 6

Write a recursive function `pp_trie` that pretty prints a trie in alphabetically sorted order by replacing common prefixes with '-' characters. Test your implementation with the following example data:

```
trie = {}
insert(trie, 'chat', 'cat')
insert(trie, 'souris', 'mouse')
insert(trie, 'chien', 'dog')
insert(trie, 'chair', 'flesh')
insert(trie, 'chic', 'stylish')
insert(trie, 'cheval','horse')
```

```
trie = dict(trie)
pprint.pprint(trie, width=40)

{'c': {'h': {'a': {'i': {'r': {'value': 'flesh'}},
                   't': {'value': 'cat'}},
             'e': {'v': {'a': {'l': {'value': 'horse'}}}},
             'i': {'c': {'value': 'stylish'},
                   'e': {'n': {'value': 'dog'}}}}},
 's': {'o': {'u': {'r': {'i': {'s': {'value': 'mouse'}}}}}}}

pp_trie(trie)

chair: flesh
---t: cat
--eval: horse
--ic: stylish
---en: dog
souris: mouse
```

Exercise 7

The *Catalan numbers* arise in many applications of combinatorial mathematics, including the counting of parse trees. The series can be defined as follows: $C_0 = 1$, and $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$ for $n \geq 0$.
Write:

a) a recursive function `cn(n)` to compute the nth Catalan number C_n,

b) a corresponding function `cn2(n)` that uses dynamic programming by storing calculated solutions in a lookup table,

c) a function `cn3(n)`, which is identical to `cn(n)` but uses a `memoize` decorator.

Test your functions first by calculating the Catalan numbers $C_0 \ldots C_{16}$ and then by using the `timeit` module:

```
print(timeit.timeit("cn(16)", setup="from __main__ import cn", number=5))
print(timeit.timeit("cn2(16)", setup="from __main__ import cn2", number=5))
print(timeit.timeit("cn3(16)", setup="from __main__ import cn3", number=5))
```

Exercise 8

Write a recursive predicate in SWI-Prolog to calculate Catalan numbers, which corresponds to a) from the previous exercise. Write then a second predicate in analogy to b) from above. The lookup table can be realized by asserting calculated solutions as dynamic facts, e.g. `assert(cn_fact(N, CN))`. Test both predicates by calculating C_{16}.