

# **EngSci Press Project Final Report**

Yunhao Qian

SN: 1005684225

April 8, 2020

# 1 Introduction

Hello.

## 2 Objectives

The core dictionary program should:

1. *Launch and response fast.* A slow start-up puts the user in a bad mood even before s/he starts using the program.

**Metric:** Measure the time interval between a user request and its response. Shorter time in seconds is better. Start-up should take less than 1 second.

2. *Use memory efficiently.* Users might run the program on an outdated computer or a virtual machine, which typically has very limited memory. Large memory use hurts performance and can cause system failure.

**Metric:** Measure the increased memory usage after loading the same dictionary dataset. Less memory in megabytes is better.

3. *Add dictionary entries easily.* The provided data have a lot of typos. Users like me might be unsatisfied and want to customize them. After following a clear and simple procedure, users should be able to add data files with the same format.

**Metric:** Count the number of operations to load a CSV file into the dictionary dataset. Fewer operations are better.

The story writer program should:

1. *Produce grammatically correct sentences.* To generate meaningful and logical stories is beyond my ability. To tell my story writer apart from a monkey hitting keys, the only way is to force my production grammatically correct.

**Metric:** Copy and paste the produced text into Microsoft Word. Green underlines flag grammatical errors. Fewer grammatical errors per sentence are better.

2. *Control the length of generated text accurately.* Sentence generation is slow, so it is a waste of time to work on unneeded sentences.

**Metric:** Calculate the percentage difference between the user-specified length and the length of generated text. Smaller average difference is better.

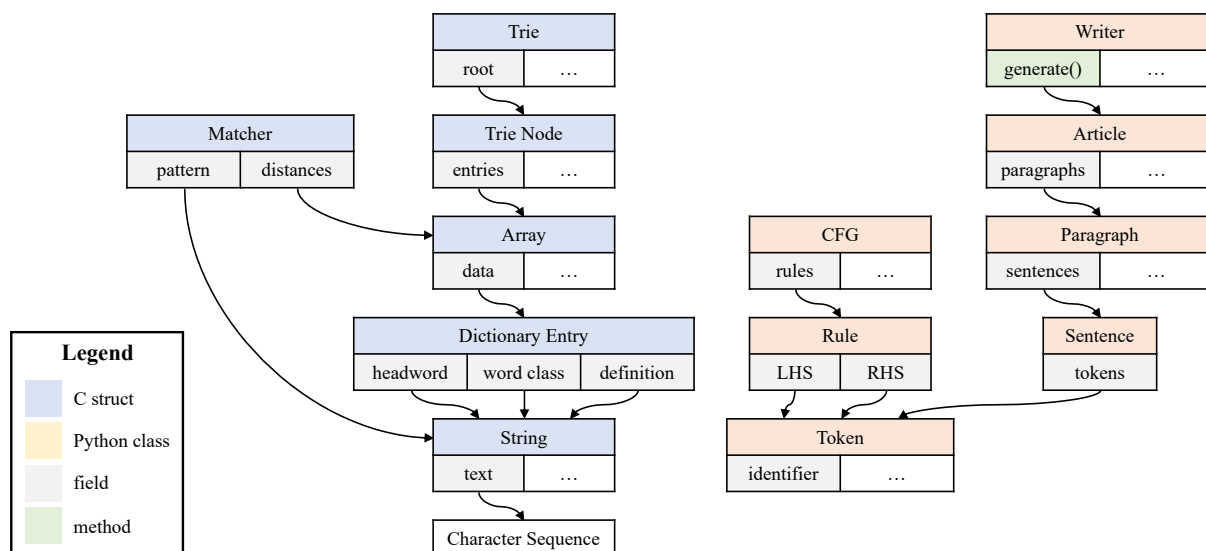


Figure 1: Overview of the project structure. Some fields and methods are omitted.

## 3 Detailed Framework

### 3.1 High-Level Overview

### 3.2 Languages

I use C for the core dictionary because it runs faster and provides more precise memory control. I initially wrote it in Python, but it took 3 seconds to launch and violated the time constraint. The bottleneck turns out to be CPU computation as opposed to disk IO. Moving to C should effectively speed it up since compiled languages typically compute much faster than interpreted languages.

I use Python for the story writer because it is easier to code, supports regular expression and features various sampling methods. Usage of these functionalities is described in <Section>. Python libraries such as NumPy have a mature and efficient C/Fortran back-end. Compared to reinvented wheels, they are faster, more robust and easier to debug. Moreover, exception mechanism in Python makes it simpler to handle special cases that appear in a natural language.

### 3.3 Data Structures

#### 3.3.1 Dynamic Array

Many functions in EngSci Press require a resizable and contiguous array. The most straightforward implementation is a block memory which is reallocated on each resize. However, frequent reallocs slow down the program. To make a trade-off between fewer reallocs and more compact storage, my custom Array type applies an exponential resizing strategy. It re-

	Size	Capacity								
<table border="1"><tr><td>a</td></tr></table>	a	1	1							
a										
<table border="1"><tr><td>a</td><td>b</td></tr></table>	a	b	2	2						
a	b									
<table border="1"><tr><td>a</td><td>b</td><td>c</td><td></td></tr></table>	a	b	c		3	4				
a	b	c								
<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td></tr></table>	a	b	c	d	4	4				
a	b	c	d							
<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td></td><td></td><td></td></tr></table>	a	b	c	d	e				5	8
a	b	c	d	e						
<table border="1"><tr><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td></td><td></td></tr></table>	a	b	c	d	e	f			6	8
a	b	c	d	e	f					

Figure 2: A dynamic array reserves space for future expansion.

serves more memory than its actual size. As shown in Figure 2, the memory space doubles when  $\text{size} \geq \text{capacity}$  and halves when  $\text{size} \leq \text{capacity}/2$ . A similar strategy is used for the `String` type.

For convenience, an `Array` of pointers is designed to hold an optional destructor and execute it upon every element deletion. The destructor function frees all the memory that an element uses, both directly and indirectly.

### 3.3.2 Trie

I choose trie to store, access and modify dictionary data because it is efficient and easy to implement. Trie is a tree-like data structure that implements mapping with string keys. As shown in Figure 3, each node holds a single character. The key of a node is represented by the character sequence along the root-node path.

Headwords are lower-cased as keys of dictionary entries, enabling case-insensitive search. Moreover, keys accept only characters whose ASCII codes fall in 32–64 or 97–122, because others are either upper-cased, or meaningless to appear in a dictionary headword. As a result, a trie node has 59 children at most.

For simplicity, mapping from a node to its children is implemented with a 59-element array of ordered child pointers. Fill `NULL` if a child does not exist. Such primitive implementation seems to hurt performance at first glance, as one has to check for many null pointers. However, because the accepted character set is small, a more advanced data structure, such as BST, usually brings more overhead as opposed to efficiency.



# Appendix A Complete Code

## Contents

A.1	core/global.h . . . . .	5
A.2	core/global.c . . . . .	6
A.3	core/array.h . . . . .	6
A.4	core/global.c . . . . .	7
A.5	core/string.h . . . . .	9
A.6	core/string.c . . . . .	10
A.7	core/dict_entry.h . . . . .	15
A.8	core/dict_entry.c . . . . .	15
A.9	core/trie_node.h . . . . .	18
A.10	core/trie_node.c . . . . .	18
A.11	core/trie.h . . . . .	20
A.12	core/trie.c . . . . .	21
A.13	core/matcher.h . . . . .	25
A.14	core/matcher.c . . . . .	25
A.15	main/api.h . . . . .	29
A.16	main/api.c . . . . .	30
A.17	main/utility.h . . . . .	38
A.18	main/utility.c . . . . .	39
A.19	main/main.c . . . . .	40
A.20	writer/grammar.py . . . . .	40
A.21	writer/lexicon.py . . . . .	45
A.22	writer/espg_base.txt . . . . .	53
A.23	writer/writer.py . . . . .	57
A.24	writer/main.py . . . . .	60

## A.1 core/global.h

```
1 #ifndef CORE_GLOBAL_H_
2 #define CORE_GLOBAL_H_
3
4 #include <assert.h>
5 #include <ctype.h>
6 #include <stdbool.h>
7 #include <stdio.h>
8 #include <stdlib.h>
```

```

9 #include <string.h>
10
11 #define MAX_LINE_SIZE 3000
12 #define ALPHABET_SIZE 59
13
14 #ifndef SUPPRESS_WARNINGS
15 #define WARN(...)
16 #else
17 #define WARN(...) fprintf(stderr, __VA_ARGS__)
18 #endif // SUPPRESS_WARNINGS
19
20 typedef void (*Destructor)(void *);
21
22 bool is_valid_key_char(char c, bool case_sensitive);
23 int char_to_index(char c);
24
25 #endif // CORE_GLOBAL_H_

```

## A.2 core/global.c

```

1 #include "core/global.h"
2
3 bool is_valid_key_char(char c, bool case_sensitive) {
4     if (!case_sensitive && c >= 65 && c <= 90) {
5         return true;
6     }
7     return c >= 32 && c <= 64 || c >= 97 && c <= 122;
8 }
9
10 int char_to_index(char c) {
11     assert(is_valid_key_char(c, true) && "char_to_index: invalid character"
12 );
13     if (c < 96) {
14         return c - 32;
15     } else {
16         return c - 64;
17     }
18 }

```

## A.3 core/array.h

```

1 #ifndef CORE_ARRAY_H_
2 #define CORE_ARRAY_H_
3
4 #include "core/global.h"

```

```

5
6 typedef struct Array {
7     void **data, **head;
8     int size, capacity;
9     Destructor destructor;
10 } Array;
11
12 Array *new_array(Destructor destructor);
13 void delete_array(void *array);
14
15 void array_append(Array *array, void *value);
16 void array_remove(Array *array, int index);
17
18 void array_reserve(Array *array, int capacity);
19
20 #endif // CORE_ARRAY_H_

```

## A.4 core/global.c

```

1 #include "core/array.h"
2
3 Array *new_array(Destructor destructor) {
4     Array *array = malloc(sizeof(Array));
5     array->size = array->capacity = 0;
6     array->destructor = destructor;
7     return array;
8 }
9
10 void delete_array(void *p) {
11     Array *array = p;
12     if (array->size > 0) {
13         if (array->destructor) {
14             for (int i = 0; i < array->size; ++i) {
15                 array->destructor(array->data[i]);
16             }
17         }
18         free(array->head);
19     }
20     free(array);
21 }
22
23 static void expand_array(Array *array) {
24     if (array->capacity <= 0) {
25         array->data = array->head = malloc(sizeof(void *));
26         array->capacity = 1;

```



```

27     return;
28 }
29 size_t block_size = array->size * sizeof(void *);
30 if (array->size < array->capacity) {
31     if (array->data + array->size >= array->head + array->capacity) {
32         memmove(array->head, array->data, block_size);
33     }
34 } else {
35     array->data = array->head = realloc(array->head, 2 * block_size);
36     array->capacity *= 2;
37 }
38 }
39
40 void array_append(Array *array, void *value) {
41     expand_array(array);
42     array->data[array->size] = value;
43     ++array->size;
44 }
45
46 static void shrink_array(Array *array) {
47     if (array->size <= 0) {
48         free(array->head);
49         array->capacity = 0;
50         return;
51     }
52     if (array->size > array->capacity / 2) {
53         return;
54     }
55     size_t block_size = array->size * sizeof(void *);
56     void **head;
57     if (array->data == array->head) {
58         head = realloc(array->head, block_size);
59     } else {
60         head = malloc(block_size);
61         memcpy(head, array->data, block_size);
62         free(array->head);
63     }
64     array->data = array->head = head;
65     array->capacity = array->size;
66 }
67
68 void array_remove(Array *array, int index) {
69     assert(index >= 0 && index < array->size &&
70         "array_remove: index out of range");
71     if (array->destructor) {

```

```

72     array->destructor(array->data[index]);
73 }
74 size_t move_size;
75 if (index < array->size / 2) {
76     move_size = index * sizeof(void *);
77     memmove(array->data + 1, array->data, move_size);
78     ++array->data;
79 } else {
80     move_size = (array->size - index - 1) * sizeof(void *);
81     memmove(array->data + index, array->data + (index + 1), move_size);
82 }
83 --array->size;
84 shrink_array(array);
85 }
86
87 void array_reserve(Array *array, int capacity) {
88     assert(capacity >= array->size &&
89           "array_reserve: capacity smaller than array size");
90     if (capacity == array->capacity) {
91         return;
92     }
93     size_t reserve_size = capacity * sizeof(void *);
94     void **head;
95     if (array->size <= 0) {
96         if (array->capacity <= 0) {
97             head = malloc(reserve_size);
98         } else {
99             head = realloc(array->head, reserve_size);
100        }
101    } else if (array->data == array->head) {
102        head = realloc(array->head, reserve_size);
103    } else {
104        head = malloc(reserve_size);
105        memcpy(head, array->data, array->size * sizeof(void *));
106        free(array->head);
107    }
108    array->data = array->head = head;
109    array->capacity = capacity;
110 }

```

## A.5 core/string.h

```

1 #ifndef CORE_STRING_H_
2 #define CORE_STRING_H_
3

```

```

4 #include "core/array.h"
5
6 typedef struct String {
7     char *text, *head;
8     int size, capacity;
9 } String;
10
11 String *new_string(const char *text, int size);
12 void delete_string(void *string);
13
14 bool is_valid_key(const String *string, bool case_sensitive);
15 int string_index(const String *string, char value);
16
17 String *to_lower(const String *string);
18 void to_lower_in_place(String *string);
19
20 String *substring(const String *string, int start, int end);
21 void substring_in_place(String *string, int start, int end);
22
23 String *trim(const String *string);
24 void trim_in_place(String *string);
25
26 String *get_line(FILE *stream);
27
28 Array *split_string(const String *string);
29 String *join_strings(const Array *strings, char c);
30
31 bool string_start_with(const String *string, const char *prefix);
32
33 #endif // CORE_STRING_H_

```

## A.6 core/string.c

```

1 #include "core/string.h"
2
3 String *new_string(const char *text, int size) {
4     if (size < 0) {
5         size = strlen(text);
6     } else {
7         assert(size <= strlen(text) &&
8             "new_string: size larger than text length");
9     }
10    String *string = malloc(sizeof(String));
11    string->text = string->head = malloc((size + 1) * sizeof(char));
12    memcpy(string->text, text, size * sizeof(char));

```

```

13     string->text[size] = '\0';
14     string->size = string->capacity = size;
15     return string;
16 }
17
18 void delete_string(void *p) {
19     String *string = p;
20     free(string->head);
21     free(string);
22 }
23
24 bool is_valid_key(const String *string, bool case_sensitive) {
25     if (string->size <= 0 || string_start_with(string, "--")) {
26         return false;
27     }
28     for (int i = 0; i < string->size; ++i) {
29         if (!is_valid_key_char(string->text[i], case_sensitive)) {
30             return false;
31         }
32     }
33     return true;
34 }
35
36 int string_index(const String *string, char value) {
37     char *find = strchr(string->text, value);
38     if (!find) {
39         return -1;
40     } else {
41         return find - string->text;
42     }
43 }
44
45 String *to_lower(const String *string) {
46     String *lower = malloc(sizeof(String));
47     lower->text = lower->head = malloc((string->size + 1) * sizeof(char));
48     for (int i = 0; i <= string->size; ++i) {
49         lower->text[i] = tolower(string->text[i]);
50     }
51     lower->size = lower->capacity = string->size;
52     return lower;
53 }
54
55 void to_lower_in_place(String *string) {
56     for (int i = 0; i < string->size; ++i) {
57         string->text[i] = tolower(string->text[i]);

```

```

58     }
59 }
60
61 static void shrink_string(String *string) {
62     if (string->size > string->capacity / 2) {
63         return;
64     }
65     size_t block_size = (string->size + 1) * sizeof(char);
66     char *head;
67     if (string->text == string->head) {
68         head = realloc(string->head, block_size);
69     } else {
70         head = malloc(block_size);
71         memcpy(head, string->text, block_size);
72         free(string->head);
73     }
74     string->text = string->head = head;
75     string->capacity = string->size;
76 }
77
78 String *substring(const String *string, int start, int end) {
79     if (start >= end) {
80         return new_string("", 0);
81     }
82     assert(start >= 0 && start < string->size &&
83            "substring: start index out of range");
84     assert(end >= 0 && end <= string->size &&
85            "substring: end index out of range");
86     return new_string(string->text + start, end - start);
87 }
88
89 void substring_in_place(String *string, int start, int end) {
90     if (start == 0 && end == string->size) {
91         return;
92     }
93     if (start >= end) {
94         string->text = string->head;
95         string->text[0] = '\0';
96         string->size = 0;
97         shrink_string(string);
98         return;
99     }
100     assert(start >= 0 && start < string->size &&
101            "substring_in_place: start index out of range");
102     assert(end >= 0 && end <= string->size &&

```

```

103         "substring_in_place: end index out of range");
104     string->text += start;
105     string->size = end - start;
106     string->text[string->size] = '\0';
107     shrink_string(string);
108 }
109
110 static void get_trim_indices(const String *string, int *start, int *end) {
111     *end = 0;
112     for (int i = string->size; i > 0; --i) {
113         if (!isspace(string->text[i - 1])) {
114             *end = i;
115             break;
116         }
117     }
118     *start = *end;
119     for (int i = 0; i < *end; ++i) {
120         if (!isspace(string->text[i])) {
121             *start = i;
122             break;
123         }
124     }
125 }
126
127 String *trim(const String *string) {
128     int start, end;
129     get_trim_indices(string, &start, &end);
130     return substring(string, start, end);
131 }
132
133 void trim_in_place(String *string) {
134     int start, end;
135     get_trim_indices(string, &start, &end);
136     substring_in_place(string, start, end);
137 }
138
139 String *get_line(FILE *stream) {
140     static char buffer[MAX_LINE_SIZE + 1];
141     if (!fgets(buffer, MAX_LINE_SIZE + 1, stream)) {
142         return NULL;
143     }
144     int size = strlen(buffer);
145     if (size > 0 && buffer[size - 1] == '\n') {
146         --size;
147     } else {

```

```

148     int c;
149     do {
150         c = getc(stream);
151     } while (c != '\n' && c != EOF);
152 }
153 if (size > 0 && buffer[size - 1] == '\r') {
154     --size;
155 }
156 return new_string(buffer, size);
157 }
158
159 Array *split_string(const String *string) {
160     Array *array = new_array(delete_string);
161     int start;
162     bool in_field = false;
163     for (int i = 0; i < string->size; ++i) {
164         if (isspace(string->text[i])) {
165             if (in_field) {
166                 array_append(array, substring(string, start, i));
167                 in_field = false;
168             }
169             } else if (!in_field) {
170                 start = i;
171                 in_field = true;
172             }
173         }
174         if (in_field) {
175             array_append(array, substring(string, start, string->size));
176         }
177         return array;
178     }
179
180 String *join_strings(const Array *strings, char c) {
181     assert(strings->size > 0 && "join_strings: string list is empty");
182     int size = strings->size - 1;
183     for (int i = 0; i < strings->size; ++i) {
184         size += ((String *)strings->data[i])->size;
185     }
186     String *joined = malloc(sizeof(String)), *field;
187     size_t block_size;
188     joined->text = joined->head = malloc((size + 1) * sizeof(char));
189     for (int i = 0, j = 0; i < strings->size; ++i) {
190         if (i) {
191             joined->text[j] = c;
192             ++j;

```

```

193     }
194     field = strings->data[i];
195     block_size = field->size * sizeof(char);
196     memcpy(joined->text + j, field->text, block_size);
197     j += block_size;
198 }
199 joined->text[size] = '\0';
200 joined->size = joined->capacity = size;
201 return joined;
202 }
203
204 bool string_start_with(const String *string, const char *prefix) {
205     int size = strlen(prefix);
206     if (size > string->size) {
207         return false;
208     }
209     return !memcmp(string->text, prefix, size * sizeof(char));
210 }

```

## A.7 core/dict\_entry.h

```

1 #ifndef CORE_DICT_ENTRY_H_
2 #define CORE_DICT_ENTRY_H_
3
4 #include "core/string.h"
5
6 typedef struct DictEntry {
7     String *headword, *word_class, *definition;
8 } DictEntry;
9
10 DictEntry *new_dict_entry(const String *line);
11 void delete_dict_entry(void *entry);
12
13 bool confirm(bool default_yes, const char *message);
14 DictEntry *input_dict_entry(const String *headword);
15
16 void display_dict_entry(const DictEntry *entry);
17 void write_dict_entry(const DictEntry *entry, FILE *stream);
18
19 #endif // CORE_DICT_ENTRY_H_

```

## A.8 core/dict\_entry.c

```

1 #include "core/dict_entry.h"
2

```



```

3 DictEntry *new_dict_entry(const String *line) {
4     int open_index = string_index(line, '(');
5     if (open_index < 0) {
6         return NULL;
7     }
8     int close_index = -1;
9     for (int i = open_index + 1, depth = 1; i < line->size; ++i) {
10        if (line->text[i] == '(') {
11            ++depth;
12        } else if (line->text[i] == ')' && --depth <= 0) {
13            close_index = i;
14            break;
15        }
16    }
17    if (close_index < 0) {
18        return NULL;
19    }
20    DictEntry *entry = malloc(sizeof(DictEntry));
21    int begin_index = 0, end_index = line->size;
22    if (line->text[0] == '"' && line->text[line->size - 1] == '"') {
23        ++begin_index;
24        --end_index;
25    }
26    entry->headword = substring(line, begin_index, open_index);
27    trim_in_place(entry->headword);
28    if (!is_valid_key(entry->headword, false)) {
29        WARN("Invalid headword: %s\n", entry->headword->text);
30        delete_string(entry->headword);
31        free(entry);
32        return NULL;
33    }
34    entry->word_class = substring(line, open_index + 1, close_index);
35    entry->definition = substring(line, close_index + 1, end_index);
36    trim_in_place(entry->word_class);
37    trim_in_place(entry->definition);
38    return entry;
39 }
40
41 void delete_dict_entry(void *p) {
42     DictEntry *entry = p;
43     delete_string(entry->headword);
44     delete_string(entry->word_class);
45     delete_string(entry->definition);
46     free(entry);
47 }

```

```

48
49 bool confirm(bool default_yes, const char *message) {
50     if (message) {
51         printf("%s ", message);
52     }
53     if (default_yes) {
54         printf("([y]/n) ");
55     } else {
56         printf("(y/[n]) ");
57     }
58     bool returned;
59     String *line = get_line(stdin);
60     trim_in_place(line);
61     to_lower_in_place(line);
62     if (!strcmp(line->text, "y") || !strcmp(line->text, "yes")) {
63         returned = true;
64     } else if (!strcmp(line->text, "n") || !strcmp(line->text, "no")) {
65         returned = false;
66     } else {
67         returned = default_yes;
68     }
69     delete_string(line);
70     return returned;
71 }
72
73 DictEntry *input_dict_entry(const String *headword) {
74     if (!is_valid_key(headword, false)) {
75         WARN("Invalid headword: %s\nDo nothing.\n", headword->text);
76         return NULL;
77     }
78     DictEntry *entry = malloc(sizeof(DictEntry));
79     entry->headword = trim(headword);
80     printf("Word class: ");
81     entry->word_class = get_line(stdin);
82     printf("Definition: ");
83     entry->definition = get_line(stdin);
84     trim_in_place(entry->word_class);
85     trim_in_place(entry->definition);
86     printf("Will create the following entry:\n");
87     display_dict_entry(entry);
88     if (!confirm(true, "Continue?")) {
89         delete_dict_entry(entry);
90         printf("Do nothing.\n");
91         return NULL;
92     } else {

```

```

93     return entry;
94 }
95 }
96
97 void display_dict_entry(const DictEntry *entry) {
98     printf("%s\n%s\n%s\n", entry->headword->text, entry->word_class->text,
99         entry->definition->text);
100 }
101
102 void write_dict_entry(const DictEntry *entry, FILE *stream) {
103     fprintf(stream, "%s (%s) %s\n", entry->headword->text,
104         entry->word_class->text, entry->definition->text);
105 }

```

## A.9 core/trie\_node.h

```

1  #ifndef CORE_TRIE_NODE_H_
2  #define CORE_TRIE_NODE_H_
3
4  #include "core/dict_entry.h"
5
6  typedef struct TrieNode {
7      char letter;
8      Array *entries;
9      struct TrieNode *parent, *children[ALPHABET_SIZE];
10     int child_count;
11 } TrieNode;
12
13 TrieNode *new_trie_node(char letter, TrieNode *parent);
14 void delete_trie_node(void *node);
15
16 TrieNode *previous_trie_node(const TrieNode *node);
17 TrieNode *next_trie_node(const TrieNode *node);
18
19 TrieNode *trie_node_add_child(TrieNode *node, char letter);
20 void trie_node_remove_child(TrieNode *node, char letter);
21
22 #endif // CORE_TRIE_NODE_H_

```

## A.10 core/trie\_node.c

```

1  #include "core/trie_node.h"
2
3  TrieNode *new_trie_node(char letter, TrieNode *parent) {
4      assert((!parent || is_valid_key_char(letter, true)) &&

```

```

5         "new_trie_node: invalid character");
6     TrieNode *node = malloc(sizeof(TrieNode));
7     node->letter = letter;
8     node->entries = new_array(delete_dict_entry);
9     node->parent = parent;
10    for (int i = 0; i < ALPHABET_SIZE; ++i) {
11        node->children[i] = NULL;
12    }
13    node->child_count = 0;
14    return node;
15 }
16
17 void delete_trie_node(void *p) {
18     TrieNode *node = p;
19     delete_array(node->entries);
20     free(node);
21 }
22
23 static TrieNode *max_trie_leaf(TrieNode *node) {
24     if (node->child_count <= 0) {
25         return node;
26     }
27     for (int i = ALPHABET_SIZE - 1; i >= 0; --i) {
28         if (node->children[i]) {
29             return max_trie_leaf(node->children[i]);
30         }
31     }
32     assert(false && "max_trie_leaf: unreachable code");
33     return NULL;
34 }
35
36 TrieNode *previous_trie_node(const TrieNode *node) {
37     TrieNode *parent = node->parent;
38     if (!parent) {
39         return NULL;
40     }
41     for (int i = char_to_index(node->letter) - 1; i >= 0; --i) {
42         if (parent->children[i]) {
43             return max_trie_leaf(parent->children[i]);
44         }
45     }
46     return parent;
47 }
48
49 TrieNode *next_trie_node(const TrieNode *node) {

```

```

50     if (node->child_count > 0) {
51         for (int i = 0; i < ALPHABET_SIZE; ++i) {
52             if (node->children[i]) {
53                 return node->children[i];
54             }
55         }
56     }
57     TrieNode *parent = node->parent;
58     int index;
59     while (parent) {
60         index = char_to_index(node->letter);
61         for (int i = index + 1; i < ALPHABET_SIZE; ++i) {
62             if (parent->children[i]) {
63                 return parent->children[i];
64             }
65         }
66         node = parent;
67         parent = node->parent;
68     }
69     return NULL;
70 }
71
72 TrieNode *trie_node_add_child(TrieNode *node, char letter) {
73     int index = char_to_index(letter);
74     assert(!node->children[index] &&
75           "trie_node_add_child: child already exists");
76     node->children[index] = new_trie_node(letter, node);
77     ++node->child_count;
78     return node->children[index];
79 }
80
81 void trie_node_remove_child(TrieNode *node, char letter) {
82     int index = char_to_index(letter);
83     assert(node->child_count > 0 && node->children[index] &&
84           "trie_node_remove_child: child does not exist");
85     assert(node->children[index]->child_count <= 0 &&
86           "trie_node_remove_child: remove a non-leaf child");
87     delete_trie_node(node->children[index]);
88     node->children[index] = NULL;
89     --node->child_count;
90 }

```

## A.11 core/trie.h

```

1 #ifndef CORE_TRIE_H_

```

```

2 #define CORE_TRIE_H_
3
4 #include "core/trie_node.h"
5
6 typedef struct Trie {
7     TrieNode *root;
8     int size;
9 } Trie;
10
11 Trie *new_trie();
12 void delete_trie(void *trie);
13
14 Array *trie_search(const Trie *trie, const String *word, bool
    case_sensitive);
15 void trie_insert(Trie *trie, DictEntry *entry);
16 void trie_remove(Trie *trie, const String *word, bool case_sensitive);
17
18 String *trie_predecessor(const Trie *trie, const String *word);
19 String *trie_successor(const Trie *trie, const String *word);
20
21 Array *traverse_trie(const Trie *trie);
22
23 #endif // CORE_TRIE_H_

```

## A.12 core/trie.c

```

1 #include "core/trie.h"
2
3 Trie *new_trie() {
4     Trie *trie = malloc(sizeof(Trie));
5     trie->root = new_trie_node('\0', NULL);
6     trie->size = 0;
7     return trie;
8 }
9
10 static void clear_trie_nodes(TrieNode *root) {
11     assert(root && "clear_trie_nodes: root node is null");
12     if (root->child_count > 0) {
13         for (int i = 0; i < ALPHABET_SIZE; ++i) {
14             if (root->children[i]) {
15                 clear_trie_nodes(root->children[i]);
16             }
17         }
18     }
19     delete_trie_node(root);

```

```

20 }
21
22 void delete_trie(void *p) {
23     Trie *trie = p;
24     clear_trie_nodes(trie->root);
25     free(trie);
26 }
27
28 static TrieNode *get_trie_node(const Trie *trie, const String *word) {
29     TrieNode *node = trie->root;
30     for (int i = 0; i < word->size; ++i) {
31         int index = char_to_index(tolower(word->text[i]));
32         node = node->children[index];
33         if (!node) {
34             return NULL;
35         }
36     }
37     return node;
38 }
39
40 Array *trie_search(const Trie *trie, const String *word, bool
    case_sensitive) {
41     Array *entries = new_array(NULL);
42     if (!is_valid_key(word, false)) {
43         WARN("Invalid headword: %s\n", word->text);
44         return entries;
45     }
46     TrieNode *node = get_trie_node(trie, word);
47     if (!node || node->entries->size <= 0) {
48         return entries;
49     }
50     if (case_sensitive) {
51         DictEntry *entry;
52         for (int i = 0; i < node->entries->size; ++i) {
53             entry = node->entries->data[i];
54             if (!strcmp(entry->headword->text, word->text)) {
55                 array_append(entries, entry);
56             }
57         }
58     } else {
59         for (int i = 0; i < node->entries->size; ++i) {
60             array_append(entries, node->entries->data[i]);
61         }
62     }
63     return entries;

```

```

64 }
65
66 void trie_insert(Trie *trie, DictEntry *entry) {
67     assert(is_valid_key(entry->headword, false) &&
68           "trie_insert: invalid entry headword");
69     String *lowered = to_lower(entry->headword);
70     TrieNode *node = trie->root;
71     for (int i = 0; i < lowered->size; ++i) {
72         int index = char_to_index(lowered->text[i]);
73         if (!node->children[index]) {
74             node = trie_node_add_child(node, lowered->text[i]);
75         } else {
76             node = node->children[index];
77         }
78     }
79     delete_string(lowered);
80     array_append(node->entries, entry);
81     ++trie->size;
82 }
83
84 void trie_remove(Trie *trie, const String *word, bool case_sensitive) {
85     assert(is_valid_key(word, false) && "trie_remove: invalid headword");
86     TrieNode *node = get_trie_node(trie, word);
87     if (!node) {
88         return;
89     }
90     if (case_sensitive) {
91         int i = 0;
92         DictEntry *entry;
93         while (i < node->entries->size) {
94             entry = node->entries->data[i];
95             if (!strcmp(entry->headword->text, word->text)) {
96                 array_remove(node->entries, i);
97                 --trie->size;
98             } else {
99                 ++i;
100             }
101         }
102     } else {
103         for (int i = node->entries->size - 1; i >= 0; --i) {
104             array_remove(node->entries, i);
105             --trie->size;
106         }
107     }
108     char letter;

```



```

109     while (node->entries->size <= 0 && node->child_count <= 0 && node->
parent) {
110         letter = node->letter;
111         node = node->parent;
112         trie_node_remove_child(node, letter);
113     }
114 }
115
116 String *trie_predecessor(const Trie *trie, const String *word) {
117     TrieNode *node = get_trie_node(trie, word);
118     if (!node) {
119         WARN("Cannot find word: %s\n", word->text);
120         return NULL;
121     }
122     node = previous_trie_node(node);
123     while (node) {
124         if (node->entries->size > 0) {
125             DictEntry *entry = node->entries->data[0];
126             return to_lower(entry->headword);
127         }
128         node = previous_trie_node(node);
129     }
130     return NULL;
131 }
132
133 String *trie_successor(const Trie *trie, const String *word) {
134     TrieNode *node = get_trie_node(trie, word);
135     if (!node) {
136         WARN("Cannot find word: %s\n", word->text);
137         return NULL;
138     }
139     node = next_trie_node(node);
140     while (node) {
141         if (node->entries->size > 0) {
142             DictEntry *entry = node->entries->data[0];
143             return to_lower(entry->headword);
144         }
145         node = next_trie_node(node);
146     }
147     return NULL;
148 }
149
150 static void traverse_trie_nodes(const TrieNode *root, Array *entries) {
151     for (int i = 0; i < root->entries->size; ++i) {
152         array_append(entries, root->entries->data[i]);

```

```

153     }
154     if (root->child_count > 0) {
155         for (int i = 0; i < ALPHABET_SIZE; ++i) {
156             if (root->children[i]) {
157                 traverse_trie_nodes(root->children[i], entries);
158             }
159         }
160     }
161 }
162
163 Array *traverse_trie(const Trie *trie) {
164     Array *entries = new_array(NULL);
165     traverse_trie_nodes(trie->root, entries);
166     return entries;
167 }

```

### A.13 core/matcher.h

```

1 #ifndef CORE_TRIE_H_
2 #define CORE_TRIE_H_
3
4 #include "core/trie_node.h"
5
6 typedef struct Trie {
7     TrieNode *root;
8     int size;
9 } Trie;
10
11 Trie *new_trie();
12 void delete_trie(void *trie);
13
14 Array *trie_search(const Trie *trie, const String *word, bool
    case_sensitive);
15 void trie_insert(Trie *trie, DictEntry *entry);
16 void trie_remove(Trie *trie, const String *word, bool case_sensitive);
17
18 String *trie_predecessor(const Trie *trie, const String *word);
19 String *trie_successor(const Trie *trie, const String *word);
20
21 Array *traverse_trie(const Trie *trie);
22
23 #endif // CORE_TRIE_H_

```

### A.14 core/matcher.c

```

1 #include "core/trie.h"

```

```

2
3 Trie *new_trie() {
4     Trie *trie = malloc(sizeof(Trie));
5     trie->root = new_trie_node('\0', NULL);
6     trie->size = 0;
7     return trie;
8 }
9
10 static void clear_trie_nodes(TrieNode *root) {
11     assert(root && "clear_trie_nodes: root node is null");
12     if (root->child_count > 0) {
13         for (int i = 0; i < ALPHABET_SIZE; ++i) {
14             if (root->children[i]) {
15                 clear_trie_nodes(root->children[i]);
16             }
17         }
18     }
19     delete_trie_node(root);
20 }
21
22 void delete_trie(void *p) {
23     Trie *trie = p;
24     clear_trie_nodes(trie->root);
25     free(trie);
26 }
27
28 static TrieNode *get_trie_node(const Trie *trie, const String *word) {
29     TrieNode *node = trie->root;
30     for (int i = 0; i < word->size; ++i) {
31         int index = char_to_index(tolower(word->text[i]));
32         node = node->children[index];
33         if (!node) {
34             return NULL;
35         }
36     }
37     return node;
38 }
39
40 Array *trie_search(const Trie *trie, const String *word, bool
    case_sensitive) {
41     Array *entries = new_array(NULL);
42     if (!is_valid_key(word, false)) {
43         WARN("Invalid headword: %s\n", word->text);
44         return entries;
45     }

```

```

46     TrieNode *node = get_trie_node(trie, word);
47     if (!node || node->entries->size <= 0) {
48         return entries;
49     }
50     if (case_sensitive) {
51         DictEntry *entry;
52         for (int i = 0; i < node->entries->size; ++i) {
53             entry = node->entries->data[i];
54             if (!strcmp(entry->headword->text, word->text)) {
55                 array_append(entries, entry);
56             }
57         }
58     } else {
59         for (int i = 0; i < node->entries->size; ++i) {
60             array_append(entries, node->entries->data[i]);
61         }
62     }
63     return entries;
64 }
65
66 void trie_insert(Trie *trie, DictEntry *entry) {
67     assert(is_valid_key(entry->headword, false) &&
68           "trie_insert: invalid entry headword");
69     String *lowered = to_lower(entry->headword);
70     TrieNode *node = trie->root;
71     for (int i = 0; i < lowered->size; ++i) {
72         int index = char_to_index(lowered->text[i]);
73         if (!node->children[index]) {
74             node = trie_node_add_child(node, lowered->text[i]);
75         } else {
76             node = node->children[index];
77         }
78     }
79     delete_string(lowered);
80     array_append(node->entries, entry);
81     ++trie->size;
82 }
83
84 void trie_remove(Trie *trie, const String *word, bool case_sensitive) {
85     assert(is_valid_key(word, false) && "trie_remove: invalid headword");
86     TrieNode *node = get_trie_node(trie, word);
87     if (!node) {
88         return;
89     }
90     if (case_sensitive) {

```

```

91     int i = 0;
92     DictEntry *entry;
93     while (i < node->entries->size) {
94         entry = node->entries->data[i];
95         if (!strcmp(entry->headword->text, word->text)) {
96             array_remove(node->entries, i);
97             --trie->size;
98         } else {
99             ++i;
100         }
101     }
102 } else {
103     for (int i = node->entries->size - 1; i >= 0; --i) {
104         array_remove(node->entries, i);
105         --trie->size;
106     }
107 }
108 char letter;
109 while (node->entries->size <= 0 && node->child_count <= 0 && node->
parent) {
110     letter = node->letter;
111     node = node->parent;
112     trie_node_remove_child(node, letter);
113 }
114 }
115
116 String *trie_predecessor(const Trie *trie, const String *word) {
117     TrieNode *node = get_trie_node(trie, word);
118     if (!node) {
119         WARN("Cannot find word: %s\n", word->text);
120         return NULL;
121     }
122     node = previous_trie_node(node);
123     while (node) {
124         if (node->entries->size > 0) {
125             DictEntry *entry = node->entries->data[0];
126             return to_lower(entry->headword);
127         }
128         node = previous_trie_node(node);
129     }
130     return NULL;
131 }
132
133 String *trie_successor(const Trie *trie, const String *word) {
134     TrieNode *node = get_trie_node(trie, word);

```

```

135     if (!node) {
136         WARN("Cannot find word: %s\n", word->text);
137         return NULL;
138     }
139     node = next_trie_node(node);
140     while (node) {
141         if (node->entries->size > 0) {
142             DictEntry *entry = node->entries->data[0];
143             return to_lower(entry->headword);
144         }
145         node = next_trie_node(node);
146     }
147     return NULL;
148 }
149
150 static void traverse_trie_nodes(const TrieNode *root, Array *entries) {
151     for (int i = 0; i < root->entries->size; ++i) {
152         array_append(entries, root->entries->data[i]);
153     }
154     if (root->child_count > 0) {
155         for (int i = 0; i < ALPHABET_SIZE; ++i) {
156             if (root->children[i]) {
157                 traverse_trie_nodes(root->children[i], entries);
158             }
159         }
160     }
161 }
162
163 Array *traverse_trie(const Trie *trie) {
164     Array *entries = new_array(NULL);
165     traverse_trie_nodes(trie->root, entries);
166     return entries;
167 }

```

## A.15 main/api.h

```

1 #ifndef MAIN_API_H_
2 #define MAIN_API_H_
3
4 #include "core/array.h"
5 #include "core/dict_entry.h"
6 #include "core/matcher.h"
7 #include "core/string.h"
8 #include "core/trie.h"
9 #include "core/trie_node.h"

```

```

10
11 #ifndef ESP_RC_PATH
12 #define ESP_RC_PATH "../.esp_rc"
13 #endif // ESP_RC_PATH
14
15 typedef enum EspMode {
16     ESP_MODE_INTERACTIVE,
17     ESP_MODE_BACKGROUND,
18     ESP_MODE_COMMAND_LINE
19 } EspMode;
20
21 void esp_initialize(EspMode mode);
22 void esp_cleanup(EspMode mode);
23
24 bool esp_parse_arguments(Array *arguments, EspMode mode);
25
26 void esp_on_load(Array *arguments, EspMode mode);
27 void esp_on_search(Array *arguments, EspMode mode);
28 void esp_on_insert(Array *arguments, EspMode mode);
29 void esp_on_remove(Array *arguments, EspMode mode);
30 void esp_on_neighbour(Array *arguments, EspMode mode);
31 void esp_on_prefix(Array *arguments, EspMode mode);
32 void esp_on_match(Array *arguments, EspMode mode);
33 void esp_on_size(Array *arguments, EspMode mode);
34 void esp_on_save(Array *arguments, EspMode mode);
35 bool esp_on_exit(Array *arguments, EspMode mode);
36
37 #endif // MAIN_API_H_

```

## A.16 main/api.c

```

1 #include "main/api.h"
2 #include "main/utility.h"
3
4 Trie *dictionary;
5
6 void esp_initialize(EspMode mode) {
7     if (mode == ESP_MODE_INTERACTIVE) {
8         printf("
9             "                || EngSci Press Dictionary by Yunhao Qian
10            ||\n"
11            "
12            *===== \n"
13            "\nStarting...\n\n");

```

```

12     }
13     dictionary = new_trie();
14     FILE *stream = fopen(ESP_RC_PATH, "r");
15     if (stream) {
16         String *line = get_line(stream);
17         Array *arguments;
18         while (line) {
19             arguments = split_string(line);
20             esp_parse_arguments(arguments, ESP_MODE_BACKGROUND);
21             delete_array(arguments);
22             delete_string(line);
23             line = get_line(stream);
24         }
25         fclose(stream);
26     }
27 }
28
29 void esp_cleanup(EspMode mode) {
30     if (mode == ESP_MODE_INTERACTIVE) {
31         printf("\nExiting...\n");
32     }
33     delete_trie(dictionary);
34 }
35
36 bool esp_parse_arguments(Array *arguments, EspMode mode) {
37     if (arguments->size <= 0) {
38         return true;
39     }
40     String *leading = to_lower(arguments->data[0]);
41     array_remove(arguments, 0);
42     bool returned = true;
43     if (!strcmp(leading->text, "load")) {
44         esp_on_load(arguments, mode);
45     } else if (!strcmp(leading->text, "search")) {
46         esp_on_search(arguments, mode);
47     } else if (!strcmp(leading->text, "insert")) {
48         esp_on_insert(arguments, mode);
49     } else if (!strcmp(leading->text, "remove")) {
50         esp_on_remove(arguments, mode);
51     } else if (!strcmp(leading->text, "neighbour")) {
52         esp_on_neighbour(arguments, mode);
53     } else if (!strcmp(leading->text, "prefix")) {
54         esp_on_prefix(arguments, mode);
55     } else if (!strcmp(leading->text, "match")) {
56         esp_on_match(arguments, mode);

```



```

57     } else if (!strcmp(leading->text, "size")) {
58         esp_on_size(arguments, mode);
59     } else if (!strcmp(leading->text, "save")) {
60         esp_on_save(arguments, mode);
61     } else if (!strcmp(leading->text, "exit")) {
62         returned = esp_on_exit(arguments, mode);
63     } else {
64         WARN("Unknown leading argument: %s\n", leading->text);
65     }
66     delete_string(leading);
67     return returned;
68 }
69
70 void esp_on_load(Array *arguments, EspMode mode) {
71     if (mode == ESP_MODE_COMMAND_LINE) {
72         WARN_NOT_SUPPORTED("load", "command-line");
73         return;
74     }
75     if (arguments->size < 1) {
76         WARN_MISSING("file name");
77         return;
78     }
79     if (arguments->size > 1) {
80         WARN_REDUNDANT(arguments, 1);
81     }
82     const char *file_name = ((String *)arguments->data[0])->text;
83     FILE *stream = fopen(file_name, "r");
84     if (!stream) {
85         WARN("Cannot open file: %s\nDo nothing.\n", file_name);
86         return;
87     }
88     String *line = get_line(stream);
89     DictEntry *entry;
90     int count = 0;
91     while (line) {
92         if (line->size > 0) {
93             entry = new_dict_entry(line);
94             if (!entry) {
95                 WARN("Failed to parse the following line in %s:\n%s\n",
96                     file_name, line->text);
97             } else {
98                 trie_insert(dictionary, entry);
99                 ++count;
100             }
101         }

```

```

102     delete_string(line);
103     line = get_line(stream);
104 }
105 fclose(stream);
106 if (mode == ESP_MODE_INTERACTIVE) {
107     printf("%d entries loaded from %s\n", count, file_name);
108 }
109 }
110
111 void esp_on_search(Array *arguments, EspMode mode) {
112     if (mode == ESP_MODE_BACKGROUND) {
113         WARN_NOT_SUPPORTED("search", "background");
114         return;
115     }
116     if (arguments->size <= 0) {
117         WARN_MISSING("headword");
118         return;
119     }
120     String *word = join_strings(arguments, ' ');
121     bool case_sensitive = false;
122     for (int i = 0; i < word->size; ++i) {
123         if (isupper(word->text[i])) {
124             case_sensitive = true;
125             break;
126         }
127     }
128     Array *results = trie_search(dictionary, word, case_sensitive);
129     if (results->size <= 0) {
130         WARN("Find no entry named: %s\n", word->text);
131         word_hint(word, dictionary, case_sensitive);
132     } else {
133         for (int i = 0; i < results->size; ++i) {
134             putchar('\n');
135             display_dict_entry(results->data[i]);
136         }
137         putchar('\n');
138     }
139     delete_array(results);
140     delete_string(word);
141 }
142
143 void esp_on_insert(Array *arguments, EspMode mode) {
144     if (mode == ESP_MODE_BACKGROUND) {
145         WARN_NOT_SUPPORTED("insert", "background");
146         return;

```

```

147     }
148     if (mode == ESP_MODE_COMMAND_LINE) {
149         WARN_NOT_SUPPORTED("insert", "command-line");
150         return;
151     }
152     if (arguments->size <= 0) {
153         WARN_MISSING("headword");
154         return;
155     }
156     String *headword = join_strings(arguments, ' ');
157     DictEntry *entry = input_dict_entry(headword);
158     if (entry) {
159         trie_insert(dictionary, entry);
160     }
161     delete_string(headword);
162 }
163
164 void esp_on_remove(Array *arguments, EspMode mode) {
165     if (mode == ESP_MODE_COMMAND_LINE) {
166         WARN_NOT_SUPPORTED("remove", "command-line");
167         return;
168     }
169     if (arguments->size <= 0) {
170         WARN_MISSING("headword");
171         return;
172     }
173     String *headword = join_strings(arguments, ' ');
174     bool case_sensitive = false;
175     for (int i = 0; i < headword->size; ++i) {
176         if (isupper(headword->text[i])) {
177             case_sensitive = true;
178             break;
179         }
180     }
181     Array *results = trie_search(dictionary, headword, case_sensitive);
182     int remove_count = results->size;
183     bool shall_remove = true;
184     if (remove_count <= 0) {
185         WARN("Find no entry named: %s\n", headword->text);
186         word_hint(headword, dictionary, case_sensitive);
187         shall_remove = false;
188     } else if (mode == ESP_MODE_INTERACTIVE) {
189         printf("The following entries will be removed:\n");
190         for (int i = 0; i < remove_count; ++i) {
191             putchar('\n');

```

```

192         display_dict_entry(results->data[i]);
193     }
194     shall_remove = confirm(true, "\nWant to continue?");
195 }
196 delete_array(results);
197 if (shall_remove) {
198     trie_remove(dictionary, headword, case_sensitive);
199     if (mode == ESP_MODE_INTERACTIVE) {
200         printf("%d entries removed.\n", remove_count);
201     }
202 } else if (mode == ESP_MODE_INTERACTIVE) {
203     printf("Do nothing.\n");
204 }
205 }
206
207 void esp_on_neighbour(Array *arguments, EspMode mode) {
208     if (mode == ESP_MODE_BACKGROUND) {
209         WARN_NOT_SUPPORTED("neighbour", "background");
210         return;
211     }
212     if (arguments->size <= 0) {
213         WARN_MISSING("headword");
214         return;
215     }
216     int radius = 10;
217     if (string_start_with(arguments->data[0], "--")) {
218         if (arguments->size <= 1) {
219             WARN_MISSING("headword");
220             return;
221         }
222         int number = parse_unsigned_int_flag(arguments->data[0]);
223         if (number < 0) {
224             WARN("Invalid flag: %s\n", ((String *)arguments->data[0])->text
225 );
226         } else {
227             radius = number;
228         }
229         array_remove(arguments, 0);
230     }
231     String *word = join_strings(arguments, ' ');
232     delete_string(word);
233 }
234 void esp_on_prefix(Array *arguments, EspMode mode) {
235     if (mode == ESP_MODE_BACKGROUND) {

```

```

236     WARN_NOT_SUPPORTED("prefix", "background");
237     return;
238 }
239 if (arguments->size <= 0) {
240     WARN_MISSING("prefix string");
241     return;
242 }
243 int max_count = 10;
244 if (string_start_with(arguments->data[0], "--")) {
245     if (arguments->size <= 1) {
246         WARN_MISSING("prefix string");
247         return;
248     }
249     int number = parse_unsigned_int_flag(arguments->data[0]);
250     if (number < 0) {
251         WARN("Invalid flag: %s\n", ((String *)arguments->data[0])>text
252 );
253     } else {
254         max_count = number;
255     }
256     array_remove(arguments, 0);
257 }
258 String *prefix = join_strings(arguments, ' ');
259 delete_string(prefix);
260 }
261 void esp_on_match(Array *arguments, EspMode mode) {
262     if (mode == ESP_MODE_BACKGROUND) {
263         WARN_NOT_SUPPORTED("match", "background");
264         return;
265     }
266     if (arguments->size <= 0) {
267         WARN_MISSING("headword");
268         return;
269     }
270     int tolerance = -1;
271     if (string_start_with(arguments->data[0], "--")) {
272         if (arguments->size <= 1) {
273             WARN_MISSING("headword");
274             return;
275         }
276         int number = parse_unsigned_int_flag(arguments->data[0]);
277         if (number < 0) {
278             WARN("Invalid flag: %s\n", ((String *)arguments->data[0])>text
279 );

```

```

279     } else {
280         tolerance = number;
281     }
282     array_remove(arguments, 0);
283 }
284 String *pattern = join_strings(arguments, ' ');
285 String *matched = trie_closest_match(dictionary, pattern, tolerance);
286 if (!matched) {
287     WARN("Find no entry similar to: %s\n", pattern->text);
288 } else {
289     printf("%s\n", matched->text);
290     delete_string(matched);
291 }
292 delete_string(pattern);
293 }
294
295 void esp_on_size(Array *arguments, EspMode mode) {
296     if (mode == ESP_MODE_BACKGROUND) {
297         WARN_NOT_SUPPORTED("size", "background");
298         return;
299     }
300     if (arguments->size > 0) {
301         WARN_REDUNDANT(arguments, 0);
302     }
303     printf("Dictionary size: %d\n", dictionary->size);
304 }
305
306 void esp_on_save(Array *arguments, EspMode mode) {
307     if (arguments->size < 1) {
308         WARN_MISSING("file name");
309         return;
310     }
311     if (arguments->size > 1) {
312         WARN_REDUNDANT(arguments, 1);
313     }
314     if (mode == ESP_MODE_INTERACTIVE && dictionary->size <= 0 &&
315         !confirm(false, "The dictionary is empty. Continue?")) {
316         printf("Do nothing.\n");
317         return;
318     }
319     const char *file_name = ((String *)arguments->data[0])->text;
320     FILE *stream = fopen(file_name, "wb");
321     if (!stream) {
322         WARN("Cannot open file: %s\nDo nothing.\n", file_name);
323         return;

```

```

324     }
325     Array *entries = traverse_trie(dictionary);
326     for (int i = 0; i < entries->size; ++i) {
327         write_dict_entry(entries->data[i], stream);
328     }
329     delete_array(entries);
330     fclose(stream);
331     if (mode == ESP_MODE_INTERACTIVE) {
332         printf("%d entries saved to %s.\n", dictionary->size, file_name);
333     }
334 }
335
336 bool esp_on_exit(Array *arguments, EspMode mode) {
337     if (mode == ESP_MODE_BACKGROUND) {
338         WARN_NOT_SUPPORTED("exit", "background");
339         return true;
340     }
341     if (mode == ESP_MODE_COMMAND_LINE) {
342         WARN_NOT_SUPPORTED("exit", "command-line");
343         return true;
344     }
345     if (arguments->size > 0) {
346         WARN_REDUNDANT(arguments, 0);
347     }
348     return !confirm(true, "Are you sure you want to exit?");
349 }

```

## A.17 main/utility.h

```

1  #ifndef MAIN_UTILITY_H_
2  #define MAIN_UTILITY_H_
3
4  #include "main/api.h"
5
6  #define WARN_NOT_SUPPORTED(argument, mode)
7      \
8      WARN("Does not support \"%s\" in %s mode.\n", argument, mode);
9
10 #define WARN_MISSING(expected)
11     \
12     WARN("Missing argument: %s expected.\n", expected)
13
14 #define WARN_REDUNDANT(arguments, start_index)
15     \
16     WARN("Redundant arguments: ignore arguments since \"%s\".\n",

```

```

14         \
15         ((String *) (arguments)->data[start_index])->text)
16 int parse_unsigned_int_flag(const String *string);
17
18 void word_hint(const String *string, const Trie *dictionary,
19               bool case_sensitive);
20
21 #endif // MAIN_UTILITY_H_

```

## A.18 main/utility.c

```

1 #include "main/utility.h"
2
3 int parse_unsigned_int_flag(const String *string) {
4     assert(string_start_with(string, "--") &&
5            "parse_unsigned_int_flag: not a flag");
6     if (string->size <= 2) {
7         return -1;
8     }
9     String *flag = substring(string, 2, string->size);
10    for (int i = 0; i < flag->size; ++i) {
11        if (!isdigit(flag->text[i])) {
12            delete_string(flag);
13            return -1;
14        }
15    }
16    int number = atoi(flag->text);
17    delete_string(flag);
18    return number;
19 }
20
21 void word_hint(const String *string, const Trie *dictionary,
22               bool case_sensitive) {
23     String *matched = trie_closest_match(dictionary, string, -1);
24     if (!matched) {
25         return;
26     }
27     if (case_sensitive) {
28         String *lowered = to_lower(string);
29         if (!strcmp(matched->text, lowered->text)) {
30             printf("Tip: use lower-case word for case-insensitive "
31                   "search/remove.\n");
32             delete_string(lowered);
33             return;

```



```

34     }
35     delete_string(lowered);
36 }
37 printf("Did you mean: %s\n", matched->text);
38 delete_string(matched);
39 }

```

## A.19 main/main.c

```

1 #include "main/api.h"
2
3 int main(int argc, const char **argv) {
4     Array *arguments;
5     if (argc > 1) {
6         arguments = new_array(delete_string);
7         for (int i = 1; i < argc; ++i) {
8             array_append(arguments, new_string(argv[i], -1));
9         }
10        esp_initialize(ESP_MODE_COMMAND_LINE);
11        esp_parse_arguments(arguments, ESP_MODE_COMMAND_LINE);
12        delete_array(arguments);
13        esp_cleanup(ESP_MODE_COMMAND_LINE);
14        return 0;
15    }
16    esp_initialize(ESP_MODE_INTERACTIVE);
17    String *line;
18    bool shall_continue;
19    do {
20        printf(">>> ");
21        line = get_line(stdin);
22        if (!line) {
23            break;
24        }
25        arguments = split_string(line);
26        shall_continue = esp_parse_arguments(arguments,
ESP_MODE_INTERACTIVE);
27        delete_array(arguments);
28        delete_string(line);
29    } while (shall_continue);
30    esp_cleanup(ESP_MODE_INTERACTIVE);
31 }

```

## A.20 writer/grammar.py

```

1 from random import choices, random

```

```

2 from re import compile, match
3
4
5 float_pattern = compile(r'([0-9]*\.)?[0-9]+\s*:\s+')
6 token_pattern = compile(
7     r'(([\s"\(\)]|(\([^\(\)]|(\("[^\"]|\\")+\"))+))|(\("[^\"]|\\")*\")
8     +)\s*')
9
10 class Token:
11
12     __slots__ = 'identifier', 'terminal', 'optional', 'probability'
13
14     def __init__(self, string):
15         self.terminal = False
16         self.optional = False
17         self.probability = 1
18         if string.startswith('(') and string.endswith(')'):
19             string = string[1:-1]
20             matched = float_pattern.match(string)
21             factor = 1
22             if matched:
23                 string = string[matched.end():]
24                 factor = float(matched.group(1))
25             self.__init__(string)
26             self.optional = True
27             self.probability *= factor
28         elif string.startswith('"') and string.endswith('"'):
29             self.identifier = string[1:-1].replace('\\"', '"')
30             self.terminal = True
31         else:
32             self.identifier = string
33
34     def __eq__(self, other):
35         if not isinstance(other, Token):
36             return False
37         return self.identifier == other.identifier and \
38             self.terminal == other.terminal
39
40     def __hash__(self):
41         return hash((self.identifier, self.terminal))
42
43     def __str__(self):
44         string = self.identifier.replace('"', '\\"')
45         if self.terminal:

```

```

46         string = "{}".format(string)
47     if self.optional:
48         if self.probability == 1:
49             string = '{}'.format(string)
50         else:
51             string = '({}: {})'.format(self.probability, string)
52     return string
53
54
55 class Rule:
56
57     __slots__ = 'lhs', 'rhs', 'weight'
58
59     def __init__(self, string):
60         matched = float_pattern.match(string)
61         if matched:
62             string = string[matched.end():]
63             self.weight = float(matched.group(1))
64         else:
65             self.weight = 1
66         matched = token_pattern.match(string)
67         string = string[matched.end():]
68         self.lhs = Token(matched.group(1))
69         string = string[match(r'>\s*', string).end():]
70         self.rhs = []
71         while True:
72             matched = token_pattern.match(string)
73             if not matched:
74                 break
75             string = string[matched.end():]
76             self.rhs.append(Token(matched.group(1)))
77
78     def __eq__(self, other):
79         if not isinstance(other, Rule):
80             return False
81         return self.lhs, tuple(self.rhs), self.weight == \
82             other.lhs, tuple(other.rhs), other.weight
83
84     def __hash__(self):
85         return hash((self.lhs, tuple(self.rhs), self.weight))
86
87     def __str__(self):
88         elements = []
89         if self.weight != 1:
90             elements.append(str(self.weight) + ':')

```

```

91     elements += [str(self.lhs), '->']
92     for element in self.rhs:
93         elements.append(str(element))
94     return ' '.join(elements)
95
96
97 class CFG:
98
99     __slots__ = 'rules', 'convergence'
100
101     def __init__(self, string=None):
102         self.rules = {}
103         self.convergence = 1
104         if string:
105             self.load_lines(string)
106
107     def __str__(self):
108         elements = []
109         if self.convergence != 1:
110             elements.append('convergence = {}'.format(self.convergence))
111         for rule_list in self.rules.values():
112             for rule in rule_list:
113                 elements.append(str(rule))
114         return '\n'.join(elements)
115
116     def load_line(self, line):
117         line = line.strip()
118         if line == '' or line.startswith('//'):
119             return
120         matched = match(r'convergence\s*=\s*([0-9]*\.)?[0-9]+\s*', line)
121         if matched:
122             self.convergence = float(matched.group(1))
123             return
124         rule = Rule(line)
125         if rule.lhs in self.rules:
126             self.rules[rule.lhs].append(rule)
127         else:
128             self.rules[rule.lhs] = [rule]
129
130     def load_lines(self, string):
131         for line in string.split('\n'):
132             try:
133                 self.load_line(line)
134             except:
135                 print('Failed to parse line: {}'.format(line))

```

```

136
137     def generate(self, start=Token('S'), max_length=-1):
138         weight_dict = {}
139         for rule_list in self.rules.values():
140             for rule in rule_list:
141                 weight_dict[rule] = rule.weight
142         stack = [start]
143         terminals = []
144         while len(stack) > 0:
145             token = stack.pop()
146             if token.optional and random() > token.probability:
147                 continue
148             if token.terminal:
149                 terminals.append(token.identifier)
150                 continue
151             try:
152                 rule_list = self.rules[token]
153             except:
154                 raise Exception('Failed to find rule for: {}'.format(token))
155         )
156         weights = [weight_dict[rule] for rule in rule_list]
157         rule = choices(rule_list, weights, k=1)[0]
158         weight_dict[rule] *= self.convergence
159         stack += rule.rhs[::-1]
160         if max_length > 0 and len(terminals) > max_length:
161             raise Exception('Exceed max length: {}'.format(max_length))
162         return terminals
163
164 demo_grammar = '''
165 convergence = 0.3
166
167 0.9: S -> Clause "."
168 0.1: S -> Clause "while" S
169
170 Clause -> NP VP
171
172 0.9: NP -> Det (0.6: Adj) N
173 0.1: NP -> NP "and" NP
174
175 VP -> V NP
176 VP -> V
177
178 Det -> "a"
179 Det -> "the"

```

```

180
181 Adj -> "smart"
182 Adj -> "tired"
183 Adj -> "brown"
184
185 N -> "student"
186 N -> "laptop"
187 N -> "car"
188
189 V -> "drives"
190 V -> "walks"
191 V -> "leaves"
192 '''
193
194
195 if __name__ == '__main__':
196     cfg = CFG(demo_grammar)
197     print(cfg)
198     while True:
199         input('-----')
200     ')
201     try:
202         tokens = cfg.generate(max_length=30)
203     except Exception as exception:
204         print(exception)
205     else:
206         tokens[0] = tokens[0].capitalize()
207         print(' '.join(tokens[:-1]) + tokens[-1])

```

## A.21 writer/lexicon.py

```

1 from collections import defaultdict
2
3
4 class DictEntry:
5
6     __slots__ = 'headword', 'word_class'
7
8     def __init__(self, headword, word_class):
9         self.headword = headword
10        self.word_class = word_class
11
12    @staticmethod
13    def from_line(line):
14        if line.startswith('"""') and line.endswith('"""'):

```

```

15         line = line[1:-1]
16         left_index = line.index('(')
17         if left_index < 0:
18             raise Exception('missing word class', line)
19         right_index = -1
20         depth = 1
21         for i in range(left_index + 1, len(line)):
22             if line[i] == '(':
23                 depth += 1
24             elif line[i] == ')':
25                 depth -= 1
26                 if depth == 0:
27                     right_index = i
28                     break
29         if right_index < 0:
30             raise Exception('mismatched brackets', line)
31         headword = line[:left_index].strip()
32         if headword == '':
33             raise Exception('empty headword', line)
34         word_class = line[left_index + 1:right_index].strip()
35         return DictEntry(headword, word_class)
36
37
38 pos_tag_to_word_classes = {
39     'Proper-Noun-Sg': set(),
40     'Proper-Noun-Pl': set(),
41     'Noun-Sg': {
42         'n. & v.',
43         'n.& v.',
44         'n & v.',
45         'n. & v. t.',
46         'n.',
47         'n. sing & pl.',
48         'a & n.',
49         'n. & v. i.',
50         'n. /',
51         'n. / interj.',
52         'n. & v.',
53         'sing. or pl.',
54         'n.sing & pl.',
55         'n',
56         'n., a., & v.',
57         'n. & a.',
58         'sing. & pl.',
59         'n .',

```

```

60     'v. t. & n.',
61     'n. sing. & pl.',
62     'a., n., & adv.',
63     'n. & adv.',
64     'n. / v. t. & i.',
65     'n.sing. & pl.',
66     'n. .',
67     'v.& n.',
68     'n. & interj.',
69     'adv. & n.',
70     'n. Chem.',
71     'v. i. & n.',
72     'n.',
73     'sing.',
74     'N.',
75     'n./',
76     'adv., & n.',
77     'a. / n.',
78     'v. & n.',
79     'a., adv., & n.',
80     'n..',
81     'n. sing. & pl',
82     'interj. & n.',
83     'n. sing.',
84     'n. & i.',
85     'imperative sing.',
86     'syntactically sing.'
87 },
88 'Noun-Pl': {
89     'n. pl.',
90     'n. sing & pl.',
91     'n.pl.',
92     'sing. or pl.',
93     'n.sing & pl.',
94     'sing. & pl.',
95     'n. pl',
96     'n. sing. & pl.',
97     'n.sing. & pl.',
98     'n pl.',
99     'n., sing. & pl.',
100    'n. collect. & pl.',
101    'n. sing. & pl',
102    'n. pl.',
103    'sing. / pl.'
104 },

```



```

105 'Gerund-V': {
106     'p. pr. & v. n.',
107     'p. pr. &, vb. n.',
108     'imp. & p. p. Fenced (/); p. pr. & vb. n.',
109     'imp. & p. p. & vb. n.',
110     'p, pr. & vb. n.',
111     'p. pr. a. & vb. n.',
112     'p. pr. vb. n.',
113     'imp. & p. pr. & vb. n.',
114     'pr.p. & vb. n.',
115     'p. pr. / vb. n.',
116     'p]. pr. & vb. n.',
117     'p. pr.& vb. n.',
118     'p. pr. &vb. n.',
119     'p. pr. & vb/ n.',
120     'P. pr. & vb. n.',
121     'p. pr. & vvb. n.',
122     'p. a. & vb. n.',
123     'p. pr. &. vb. n.',
124     'p. pr. & pr. & vb. n.',
125     'vb. n.',
126     'p. p. & vb. n.',
127     'p pr. & vb. n.',
128     'imp. & p. p. Adored (/); p. pr. & vb. n.',
129     'p. pr & vb. n.'
130 },
131 'Verb-I-Sg': {
132     '3d sing.pr.',
133     'subj. 3d pers. sing.',
134     '3d sing.',
135     '3d pers. sing. pres.',
136     '3d sing. pr.',
137     'pres. indic. sing., 1st & 3d pers.',
138     'Sing. pres. ind.',
139     '3d sing.',
140     'pres. sing.'
141 },
142 'Verb-T-Sg': {
143     '3d sing.pr.',
144     'subj. 3d pers. sing.',
145     '3d sing.',
146     '3d pers. sing. pres.',
147     '3d sing. pr.',
148     'pres. indic. sing., 1st & 3d pers.',
149     'Sing. pres. ind.',

```

```

150     '3d sing.',
151     'pres. sing.'
152 },
153 'Verb-I-Pl': {
154     'v. t. / i.',
155     'v. i.',
156     'n. & v. i.',
157     'v. i. & i.',
158     'v. i.',
159     'v.t & i.',
160     'v.i',
161     'v. t. / v. i.',
162     'v.i.',
163     'v. t.& i.',
164     'n. / v. t. & i.',
165     'v. i.',
166     'v. t. & v. i.',
167     'v. i. & n.',
168     'v. i. & auxiliary.',
169     'v. t. & i.',
170     'v. i. & t.',
171     'v. i. / auxiliary'
172 },
173 'Verb-T-Pl': {
174     'v. t. / i.',
175     'a. & v. t.',
176     'v. t. &',
177     'n. & v. t.',
178     'v. t..',
179     'v. t. v. t.',
180     'v.t & i.',
181     'v. t. / v. i.',
182     'v.t',
183     'v. t. & n.',
184     'v. t.& i.',
185     'n. / v. t. & i.',
186     'v. t. & v. i.',
187     'v./t.',
188     'v. t.',
189     'v. t. / auxiliary',
190     'v. t.',
191     'v. i. & t.',
192     'v.t.'
193 },
194 'Preposition': {

```

```

195     'prep., adv., & conj.',
196     'prep., adv., conj. & n.',
197     'adv. & prep.',
198     'prep. & conj., but properly a participle',
199     'prep., adv. & a.',
200     'prep., adv. & conj.',
201     'prep. & adv.',
202     'adv., prep., & conj.',
203     'prep.',
204     'adv. or prep.',
205     'prep. & conj.',
206     'conj. & prep.'
207 },
208 'Adj': {
209     'adj.',
210     'pron. / adj.',
211     'a.',
212     'p. p. / a.',
213     'a. & v. t.',
214     'adv. & a.',
215     'p. p & a.',
216     'p. p. & a.',
217     'a. / a. pron.',
218     'P. p. & a.',
219     'pron. & a.',
220     'a & n.',
221     'a/',
222     'adv. / a.',
223     'a. & a. pron.',
224     'a & p. p.',
225     'p. & a.',
226     'prep., adv. & a.',
227     'a. .',
228     'a. superl.',
229     'v. & a.',
230     'a. & adv.',
231     'n., a., & v.',
232     'a. a.',
233     'pron., a., conj., & adv.',
234     'n. & a.',
235     'p. pr. a. & vb. n.',
236     'a. & v.',
237     'a., n., & adv.',
238     'a. Vigorously',
239     'a. & n.',

```

```

240     'a.',
241     'a. / adv.',
242     'a & adv.',
243     'a. Vibrating',
244     'a. or pron.',
245     'a. / pron.',
246     'imp., p. p., & a.',
247     'a',
248     'p. p. & a',
249     'a. / n.',
250     'pron., a., & adv.',
251     'a., adv., & n.',
252     'a. & p. p.',
253     'a. & pron.'
254 },
255 'Adv': {
256     'prep., adv., & conj.',
257     'prep., adv., conj. & n.',
258     'adv. & a.',
259     'adv. In combination or cooperation',
260     'adv. / interj.',
261     'interrog. adv.',
262     'adv. & prep.',
263     'adv. In a vanishing manner',
264     'adv. / a.',
265     'prep., adv. & a.',
266     'a. & adv.',
267     'pron., a., conj., & adv.',
268     'prep., adv. & conj.',
269     'conj. / adv.',
270     'adv.',
271     'prep. & adv.',
272     'interj., adv., or a.',
273     'a., n., & adv.',
274     'interj., adv., & n.',
275     'n. & adv.',
276     'a. / adv.',
277     'adv., prep., & conj.',
278     'adv. & n.',
279     'a & adv.',
280     'adv. or prep.',
281     'adv., & n.',
282     'pron., a., & adv.',
283     'a., adv., & n.',
284     'interj. & adv.',

```

```

285     'adv. / conj.',
286     'adv. & conj.'
287 }
288 }
289
290
291 def create_lexicon(lines):
292     lexicon = {}
293     for tag in pos_tag_to_word_classes:
294         lexicon[tag] = set()
295     for line in lines:
296         line = line.strip()
297         if line == '':
298             continue
299         try:
300             entry = DictEntry.from_line(line)
301         except:
302             print('Failed to parse line: {}'.format(line))
303             continue
304         for tag in pos_tag_to_word_classes:
305             word_class_set = pos_tag_to_word_classes[tag]
306             if entry.word_class in word_class_set:
307                 lexicon[tag].add(entry.headword.lower().replace("'", '\\\''))
308     )
309     return lexicon
310
311 def write_lexicon(lexicon, stream):
312     for tag in pos_tag_to_word_classes:
313         for terminal in lexicon[tag]:
314             stream.write('{} -> "{}"\n'.format(tag, terminal))
315
316
317 if __name__ == '__main__':
318     try:
319         output = open('espg_lexicon.txt', 'w')
320     except:
321         print('Failed to open file: {}'.format('espg_lexicon.txt'))
322         exit()
323     for i in range(ord('A'), ord('Z') + 1):
324         file_name = ('../Dictionary-in-csv/{}.csv'.format(chr(i)))
325         try:
326             with open(file_name, 'r') as stream:
327                 lines = open(file_name, 'r').readlines()
328         except:

```

```

329         print('Failed to open file: {}'.format(file_name))
330     else:
331         lexicon = create_lexicon(lines)
332         write_lexicon(lexicon, output)
333     output.close()

```

## A.22 writer/espg\_base.txt

```

1 // ===== Sentence =====
2 S -> NP-Sg VP-Sg "."
3 S -> NP-Pl VP-Pl "."
4 0.4: S -> VP-Pl "!"
5 0.2: S -> Aux-Sg NP-Sg VP-Pl "?"
6 0.2: S -> Aux-Pl NP-Pl VP-Pl "?"
7 0.1: S -> Wh-NP-Sg Aux-Sg NP-Sg VP-Pl "?"
8 0.1: S -> Wh-NP-Pl Aux-Pl NP-Pl VP-Pl "?"
9
10 // ===== Noun Phrase =====
11 0.2: NP-Sg -> Pronoun-Sg
12 // NP-Sg -> Proper-Noun-Sg
13 NP-Sg -> Det-Sg (0.5: AP) Nominal-Sg
14 0.2: NP-Pl -> Pronoun-Pl
15 // NP-Pl -> Proper-Noun-Pl
16 NP-Pl -> Det-Pl (0.5: AP) Nominal-Pl
17
18 // ===== Nominal =====
19 Nominal-Sg -> Noun-Sg
20 0.3: Nominal-Sg -> Nominal-Sg PP
21 0.3: Nominal-Sg -> Nominal-Sg Gerund-VP
22 0.3: Nominal-Sg -> Nominal-Sg Rel-Clause-Sg
23 Nominal-Pl -> Noun-Pl
24 0.3: Nominal-Pl -> Nominal-Pl PP
25 0.3: Nominal-Pl -> Nominal-Pl Gerund-VP
26 0.3: Nominal-Pl -> Nominal-Pl Rel-Clause-Pl
27
28 // ===== Gerundive Verb =====
29 Gerund-VP -> Gerund-V
30 Gerund-VP -> Gerund-V NP-Sg
31 Gerund-VP -> Gerund-V NP-Pl
32 Gerund-VP -> Gerund-V PP
33 Gerund-VP -> Gerund-V NP-Sg PP
34 Gerund-VP -> Gerund-V NP-Pl PP
35
36 // ===== Relative Clause =====
37 Rel-Clause-Sg -> Rel-Pronoun VP-Sg

```

```

38 Rel-Clause-Pl -> Rel-Pronoun VP-Pl
39
40 // ===== Verb Phrase =====
41 VP-Sg -> Verb-I-Sg
42 VP-Sg -> Verb-T-Sg NP-Sg
43 VP-Sg -> Verb-T-Sg NP-Pl
44 VP-Sg -> Verb-T-Sg NP-Sg PP
45 VP-Sg -> Verb-T-Sg NP-Pl PP
46 VP-Sg -> Verb-I-Sg PP
47 VP-Pl -> Verb-I-Pl
48 VP-Pl -> Verb-T-Pl NP-Sg
49 VP-Pl -> Verb-T-Pl NP-Pl
50 VP-Pl -> Verb-T-Pl NP-Sg PP
51 VP-Pl -> Verb-T-Pl NP-Pl PP
52 VP-Pl -> Verb-I-Pl PP
53
54 // ===== Adjective Phrase =====
55 AP -> Adj
56 0.2: AP -> Adv AP
57
58 // ===== Prepositional Phrase =====
59 PP -> Preposition NP-Sg
60 PP -> Preposition NP-Pl
61
62 // ===== Determiner =====
63 5: Det-Sg -> "the"
64 5: Det-Pl -> "the"
65 5: Det-Sg -> "a"
66 4: Deg-Sg -> "this"
67 4: Deg-Sg -> "that"
68 4: Det-Pl -> "these"
69 4: Det-Pl -> "those"
70 Det-Sg -> "my"
71 Det-Pl -> "my"
72 Det-Sg -> "your"
73 Det-Pl -> "your"
74 Det-Sg -> "his"
75 Det-Pl -> "his"
76 Det-Sg -> "her"
77 Deg-Pl -> "her"
78 Det-Sg -> "its"
79 Det-Pl -> "its"
80 Det-Sg -> "our"
81 Det-Pl -> "our"
82 Det-Sg -> "their"

```

```

83 Det-Pl -> "their"
84 2: Det-Pl -> "a" "few"
85 2: Det-Pl -> "many"
86 2: Det-Pl -> "a" "lot" "of"
87 3: Det-Pl -> "some"
88 Det-Sg -> "any"
89 Det-Sg -> "one"
90 Det-Pl -> "all"
91 Det-Sg -> "each"
92 Det-Sg -> "every"
93 Det-Sg -> "another"
94 Det-Sg -> NP-Sg "'s"
95 Det-Pl -> NP-Sg "'s"
96
97 // ===== Auxiliary Verb =====
98 Aux-Sg -> "has"
99 Aux-Pl -> "have"
100 Aux-Sg -> "had"
101 Aux-Pl -> "had"
102 Aux-Sg -> "did"
103 Aux-Pl -> "did"
104 Aux-Sg -> "will"
105 Aux-Pl -> "will"
106 Aux-Sg -> "should"
107 Aux-Pl -> "should"
108 Aux-Sg -> "would"
109 Aux-Pl -> "would"
110 Aux-Sg -> "may"
111 Aux-Pl -> "may"
112 Aux-Sg -> "might"
113 Aux-Pl -> "might"
114 Aux-Sg -> "must"
115 Aux-Pl -> "must"
116 Aux-Sg -> "can"
117 Aux-Pl -> "can"
118 Aux-Sg -> "could"
119 Aux-Pl -> "could"
120 Aux-Sg -> "does"
121 Aux-Pl -> "do"
122 Aux-Sg -> "need"
123 Aux-Pl -> "need"
124
125 // ===== Wh- Noun Phrase =====
126 Wh-NP-Sg -> "when"
127 Wh-NP-Pl -> "when"

```



```

128 Wh-NP-Sg -> "who"
129 Wh-NP-Pl -> "who"
130 Wh-NP-Sg -> "where"
131 Wh-NP-Pl -> "where"
132 Wh-NP-Sg -> "what"
133 Wh-NP-Pl -> "what"
134 Wh-NP-Sg -> "what" Noun-Sg
135 Wh-NP-Pl -> "what" Noun-Pl
136 Wh-NP-Sg -> "whose" Noun-Sg
137 Wh-NP-Pl -> "whose" Noun-Pl
138 Wh-NP-Sg -> "which" Noun-Sg
139 Wh-NP-Pl -> "which" Noun-Pl
140
141 // ===== Pronoun =====
142 4: Pronoun-Pl -> "you"
143 Pronoun-Sg -> "yours"
144 Pronoun-Pl -> "yours"
145 2: Pronoun-Pl -> "yourself"
146 Pronoun-Sg -> "him"
147 Pronoun-Sg -> "his"
148 Pronoun-Pl -> "his"
149 2: Pronoun-Sg -> "himself"
150 Pronoun-Sg -> "her"
151 Pronoun-Sg -> "hers"
152 Pronoun-Pl -> "hers"
153 2: Pronoun-Sg -> "herself"
154 4: Pronoun-Sg -> "it"
155 Pronoun-Sg -> "its"
156 Pronoun-Pl -> "its"
157 Pronoun-Sg -> "itself"
158 4: Pronoun-Sg -> "ours"
159 4: Pronoun-Pl -> "ours"
160 2: Pronoun-Pl -> "ourself"
161 Pronoun-Sg -> "theirs"
162 Pronoun-Pl -> "theirs"
163 2: Pronoun-Pl -> "themselves"
164
165 // ===== Relative Pronoun =====
166 Rel-Pronoun -> "who"
167 Rel-Pronoun -> "which"
168 Rel-Pronoun -> "that"
169
170 // ===== In espg_lexicon.txt =====
171 // Noun-Sg
172 // Noun-Pl

```

```

173 // Gerund-V
174 // Verb-I-Sg
175 // Verb-T-Sg
176 // Verb-I-Pl
177 // Verb-T-Pl
178 // Preposition
179 // Adj
180 // Adv

```

## A.23 writer/writer.py

```

1 from string import punctuation
2 from numpy.random import poisson
3
4
5 class Sentence:
6
7     __slots__ = 'tokens'
8
9     def __init__(self, tokens):
10         self.tokens = tokens
11
12     def __str__(self):
13         elements = []
14         for token in self.tokens:
15             if len(elements) > 0:
16                 if token[0] in punctuation or elements[-1][-1] in
punctuation:
17                     elements[-1] += token
18                 else:
19                     elements.append(token)
20             else:
21                 elements.append(token.capitalize())
22         return ' '.join(elements)
23
24     def word_count(self):
25         count = 0
26         for token in self.tokens:
27             if token not in punctuation:
28                 count += 1
29         return count
30
31
32 class Paragraph:
33

```

```

34     __slots__ = 'sentences', 'indent'
35
36     def __init__(self, indent):
37         self.sentences = []
38         self.indent = indent
39
40     def __str__(self):
41         elements = []
42         for sentence in self.sentences:
43             elements.append(str(sentence))
44         return ' ' * self.indent + ' '.join(elements)
45
46     def add_sentence(self, tokens):
47         self.sentences.append(Sentence(tokens))
48
49     def word_count(self):
50         count = 0
51         for sentence in self.sentences:
52             count += sentence.word_count()
53         return count
54
55
56 class Article:
57
58     __slots__ = 'paragraphs', 'title', 'spacing', 'indent'
59
60     def __init__(self, title, spacing, indent):
61         self.paragraphs = []
62         self.title = title
63         self.spacing = spacing
64         self.indent = indent
65
66     def __str__(self):
67         elements = []
68         if self.title:
69             elements.append(self.title)
70         for paragraph in self.paragraphs:
71             if len(paragraph.sentences) == 0:
72                 continue
73             elements.append(str(paragraph))
74         return ('\n' * (self.spacing + 1)).join(elements)
75
76     def add_sentence(self, tokens):
77         if len(self.paragraphs) == 0:
78             self.new_paragraph()

```

```

79     self.paragraphs[-1].add_sentence(tokens)
80
81     def new_paragraph(self, indent=None):
82         if len(self.paragraphs) > 0 and \
83             len(self.paragraphs[-1].sentences) == 0:
84             self.paragraphs.pop()
85         if indent == None:
86             indent = self.indent
87         self.paragraphs.append(Paragraph(indent))
88
89     def word_count(self):
90         count = 0
91         for paragraph in self.paragraphs:
92             count += paragraph.word_count()
93         return count
94
95
96 class Writer:
97
98     __slots__ = 'grammar', 'paragraphs_per_article', \
99                 'sentences_per_paragraph', 'tokens_per_sentence'
100
101     def __init__(self, grammar, paragraphs_per_article=5,
102                  sentences_per_paragraph=10, tokens_per_sentence=20):
103         self.grammar = grammar
104         self.paragraphs_per_article = paragraphs_per_article
105         self.sentences_per_paragraph = sentences_per_paragraph
106         self.tokens_per_sentence = tokens_per_sentence
107
108     def generate(self, title=None, spacing=1, indent=4):
109         article = Article(title, spacing, indent)
110         for i in range(poisson(self.paragraphs_per_article)):
111             for j in range(poisson(self.sentences_per_paragraph)):
112                 attempt_count = 0
113                 while True:
114                     try:
115                         max_length = poisson(self.tokens_per_sentence)
116                         tokens = self.grammar.generate(max_length=
max_length)
117                     except:
118                         attempt_count += 1
119                         if attempt_count > 50:
120                             raise Exception('Too many attempts')
121                     else:
122                         article.add_sentence(tokens)

```

```

123         break
124     article.new_paragraph()
125     return article
126
127
128 if __name__ == '__main__':
129     from grammar import CFG, demo_grammar
130     article = Writer(CFG(demo_grammar)).generate('Demo Article')
131     print(article)
132     print()
133     print('Word count: {}'.format(article.word_count()))

```

## A.24 writer/main.py

```

1 from grammar import CFG
2 from writer import Writer
3
4
5 def create_espg():
6     espg = CFG()
7     with open('espg_base.txt', 'r') as stream:
8         lines = stream.read()
9     espg.load_lines(lines)
10    with open('espg_lexicon.txt', 'r') as stream:
11        lines = stream.read()
12    espg.load_lines(lines)
13    return espg
14
15
16 if __name__ == '__main__':
17     espg = create_espg()
18     writer = Writer(espg, 3, 10, 15)
19     while True:
20         input('-----
21         ')
22         article = writer.generate()
23         print(article)
24         print()
25         print('Word count: {}'.format(article.word_count()))

```