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 *Use memory efficiently.* Users might run the program on an outdated computer or a virtual machine, which typically has very limited memory.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.

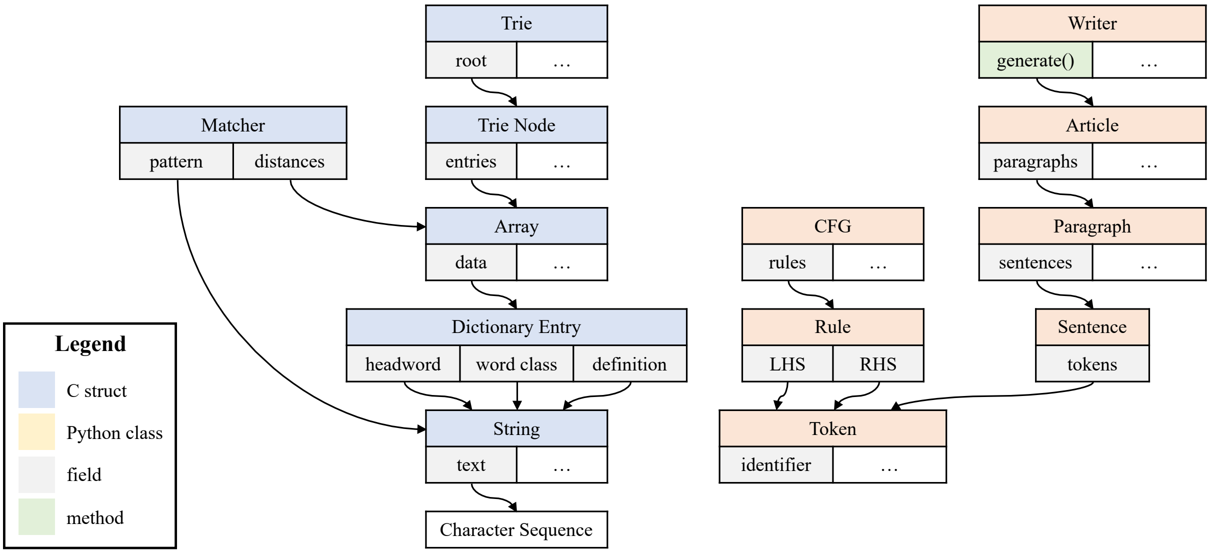

Figure 1: Overview of the project structure. Some fields and methods are omitted.

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-

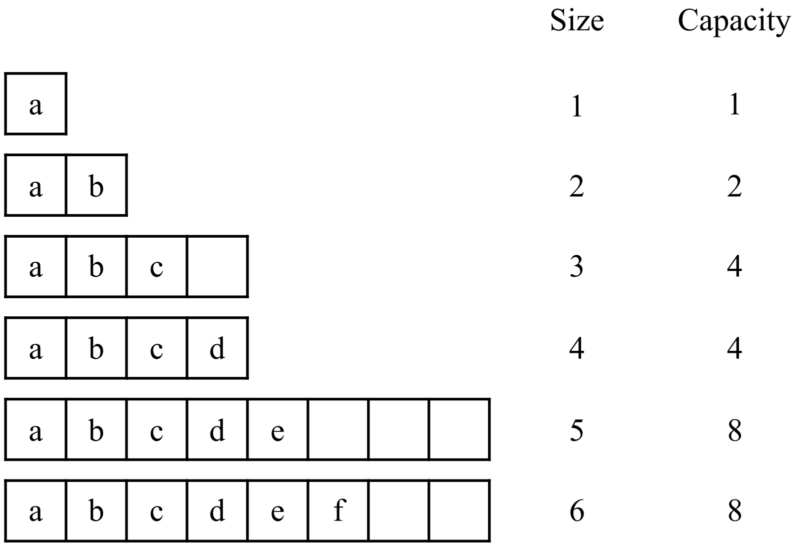

Figure 2: A dynamic array reserves space for future expansion.

serves more memory than its actual size. As shown in Figure 2, the memory space doubles when size $\geq$ capacity and halves when size $\leq$ 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.

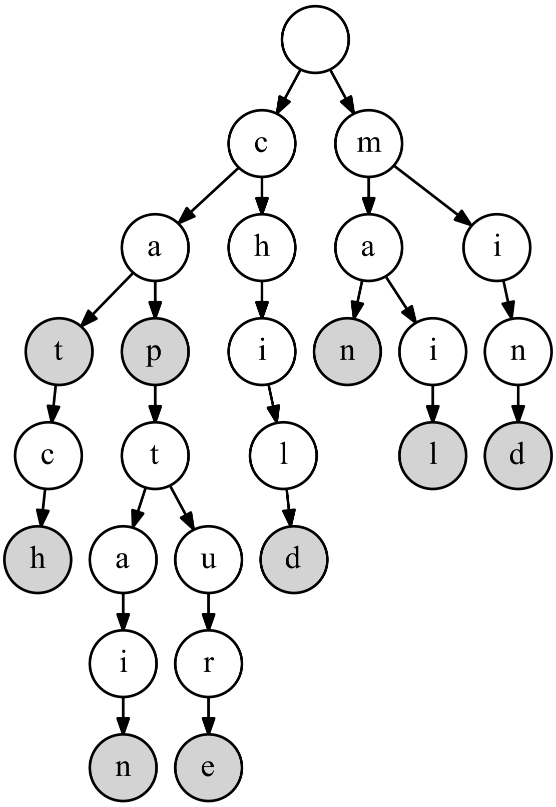

Figure 3: A trie. Each shadowed node represents an English word.

3.3.3 Levenshtein Automaton

3.4 Software Implementation

3.4.1 Context-Free Grammar

3.4.2 Story Length Control

4 Results

Hello.

5 Future Work/Conclusion

Hello.

Appendix A Complete Code

Contents

A.1 core/global.h

```c
#ifndef CORE_GLOBAL_H_
#define CORE_GLOBAL_H_

#include <assert.h>
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>

#define MAX_LINE_SIZE 3000
#define ALPHABET_SIZE 59

#ifdef SUPPRESS_WARNINGS
#define WARN(...)
#else
#define WARN(...) fprintf(stderr, __VA_ARGS__)
#endif // SUPPRESS_WARNINGS

typedef void (*Destructor)(void *);

bool is_valid_key_char(char c, bool case_sensitive);
int char_to_index(char c);

#endif // CORE_GLOBAL_H_
```

A.2 core/global.c

```c
#include "core/global.h"

bool is_valid_key_char(char c, bool case_sensitive) {
    if (!case_sensitive && c >= 65 && c <= 90) {
        return true;
    }
    return c >= 32 && c <= 64 || c >= 97 && c <= 122;
}

int char_to_index(char c) {
    assert(is_valid_key_char(c, true) && "char_to_index: invalid character"
   );
    if (c < 96) {
        return c - 32;
    } else {
        return c - 64;
    }
}
```

A.3 core/array.h

```c
#ifndef CORE_ARRAY_H_
#define CORE_ARRAY_H_

#include "core/global.h"
```

```c

typedef struct Array {
    void **data, **head;
    int size, capacity;
    Destructor destructor;
} Array;

Array *new_array(Destructor destructor);
void delete_array(void *array);

void array_append(Array *array, void *value);
void array_remove(Array *array, int index);

void array_reserve(Array *array, int capacity);

#endif // CORE_ARRAY_H_
```

A.4 core/global.c

```c
#include "core/array.h"

Array *new_array(Destructor destructor) {
    Array *array = malloc(sizeof(Array));
    array->size = array->capacity = 0;
    array->destructor = destructor;
    return array;
}

void delete_array(void *p) {
    Array *array = p;
    if (array->size > 0) {
        if (array->destructor) {
            for (int i = 0; i < array->size; ++i) {
                array->destructor(array->data[i]);
            }
        }
        free(array->head);
    }
    free(array);
}

static void expand_array(Array *array) {
    if (array->capacity <= 0) {
        array->data = array->head = malloc(sizeof(void *));
        array->capacity = 1;
        return;
    }
    size_t block_size = array->size * sizeof(void *);
    if (array->size < array->capacity) {
        if (array->data + array->size >= array->head + array->capacity) {
            memmove(array->head, array->data, block_size);
        }
    } else {
        array->data = array->head = realloc(array->head, 2 * block_size);
        array->capacity *= 2;
    }
}

void array_append(Array *array, void *value) {
    expand_array(array);
    array->data[array->size] = value;
    ++array->size;
}

static void shrink_array(Array *array) {
    if (array->size <= 0) {
        free(array->head);
        array->capacity = 0;
        return;
    }
    if (array->size > array->capacity / 2) {
        return;
    }
    size_t block_size = array->size * sizeof(void *);
    void **head;
    if (array->data == array->head) {
        head = realloc(array->head, block_size);
    } else {
        head = malloc(block_size);
        memcpy(head, array->data, block_size);
        free(array->head);
    }
    array->data = array->head = head;
    array->capacity = array->size;
}

void array_remove(Array *array, int index) {
    assert(index >= 0 && index < array->size &&
           "array_remove: index out of range");
    if (array->destructor) {
```

```c
        array->destructor(array->data[index]);
    }
    size_t move_size;
    if (index < array->size / 2) {
        move_size = index * sizeof(void *);
        memmove(array->data + 1, array->data, move_size);
        ++array->data;
    } else {
        move_size = (array->size - index - 1) * sizeof(void *);
        memmove(array->data + index, array->data + (index + 1), move_size);
    }
    --array->size;
    shrink_array(array);
}

void array_reserve(Array *array, int capacity) {
    assert(capacity >= array->size &&
           "array_reserve: capacity smaller than array size");
    if (capacity == array->capacity) {
        return;
    }
    size_t reserve_size = capacity * sizeof(void *);
    void **head;
    if (array->size <= 0) {
        if (array->capacity <= 0) {
            head = malloc(reserve_size);
        } else {
            head = realloc(array->head, reserve_size);
        }
    } else if (array->data == array->head) {
        head = realloc(array->head, reserve_size);
    } else {
        head = malloc(reserve_size);
        memcpy(head, array->data, array->size * sizeof(void *));
        free(array->head);
    }
    array->data = array->head = head;
    array->capacity = capacity;
}
```

A.5 core/string.h

```c
#ifndef CORE_STRING_H_
#define CORE_STRING_H_

```

```c
#include "core/array.h"

typedef struct String {
    char *text, *head;
    int size, capacity;
} String;

String *new_string(const char *text, int size);
void delete_string(void *string);

bool is_valid_key(const String *string, bool case_sensitive);
int string_index(const String *string, char value);

String *to_lower(const String *string);
void to_lower_in_place(String *string);

String *substring(const String *string, int start, int end);
void substring_in_place(String *string, int start, int end);

String *trim(const String *string);
void trim_in_place(String *string);

String *get_line(FILE *stream);

Array *split_string(const String *string);
String *join_strings(const Array *strings, char c);

bool string_start_with(const String *string, const char *prefix);

#endif // CORE_STRING_H_
```

A.6 core/string.c

```c
#include "core/string.h"

String *new_string(const char *text, int size) {
    if (size < 0) {
        size = strlen(text);
    } else {
        assert(size <= strlen(text) &&
                "new_string: size larger than text length");
    }
    String *string = malloc(sizeof(String));
    string->text = string->head = malloc((size + 1) * sizeof(char));
    memcpy(string->text, text, size * sizeof(char));
    string->text[size] = '\0';
    string->size = string->capacity = size;
    return string;
}

void delete_string(void *p) {
    String *string = p;
    free(string->head);
    free(string);
}

bool is_valid_key(const String *string, bool case_sensitive) {
    if (string->size <= 0 || string_start_with(string, "--")) {
        return false;
    }
    for (int i = 0; i < string->size; ++i) {
        if (!is_valid_key_char(string->text[i], case_sensitive)) {
            return false;
        }
    }
    return true;
}

int string_index(const String *string, char value) {
    char *find = strchr(string->text, value);
    if (!find) {
        return -1;
    } else {
        return find - string->text;
    }
}

String *to_lower(const String *string) {
    String *lower = malloc(sizeof(String));
    lower->text = lower->head = malloc((string->size + 1) * sizeof(char));
    for (int i = 0; i <= string->size; ++i) {
        lower->text[i] = tolower(string->text[i]);
    }
    lower->size = lower->capacity = string->size;
    return lower;
}

void to_lower_in_place(String *string) {
    for (int i = 0; i < string->size; ++i) {
        string->text[i] = tolower(string->text[i]);
    }
}

static void shrink_string(String *string) {
    if (string->size > string->capacity / 2) {
        return;
    }
    size_t block_size = (string->size + 1) * sizeof(char);
    char *head;
    if (string->text == string->head) {
        head = realloc(string->head, block_size);
    } else {
        head = malloc(block_size);
        memcpy(head, string->text, block_size);
        free(string->head);
    }
    string->text = string->head = head;
    string->capacity = string->size;
}

String *substring(const String *string, int start, int end) {
    if (start >= end) {
        return new_string("", 0);
    }
    assert(start >= 0 && start < string->size &&
           "substring: start index out of range");
    assert(end >= 0 && end <= string->size &&
           "substring: end index out of range");
    return new_string(string->text + start, end - start);
}

void substring_in_place(String *string, int start, int end) {
    if (start == 0 && end == string->size) {
        return;
    }
    if (start >= end) {
        string->text = string->head;
        string->text[0] = '\0';
        string->size = 0;
        shrink_string(string);
        return;
    }
    assert(start >= 0 && start < string->size &&
           "substring_in_place: start index out of range");
    assert(end >= 0 && end <= string->size &&
           "substring_in_place: end index out of range");
    string->text += start;
    string->size = end - start;
    string->text[string->size] = '\0';
    shrink_string(string);
}

static void get_trim_indices(const String *string, int *start, int *end) {
    *end = 0;
    for (int i = string->size; i > 0; --i) {
        if (!isspace(string->text[i - 1])) {
            *end = i;
            break;
        }
    }
    *start = *end;
    for (int i = 0; i < *end; ++i) {
        if (!isspace(string->text[i])) {
            *start = i;
            break;
        }
    }
}

String *trim(const String *string) {
    int start, end;
    get_trim_indices(string, &start, &end);
    return substring(string, start, end);
}

void trim_in_place(String *string) {
    int start, end;
    get_trim_indices(string, &start, &end);
    substring_in_place(string, start, end);
}

String *get_line(FILE *stream) {
    static char buffer[MAX_LINE_SIZE + 1];
    if (!fgets(buffer, MAX_LINE_SIZE + 1, stream)) {
        return NULL;
    }
    int size = strlen(buffer);
    if (size > 0 && buffer[size - 1] == '\n') {
        --size;
    } else {
        int c;
        do {
            c = getc(stream);
        } while (c != '\n' && c != EOF);
    }
    if (size > 0 && buffer[size - 1] == '\r') {
        --size;
    }
    return new_string(buffer, size);
}

Array *split_string(const String *string) {
    Array *array = new_array(delete_string);
    int start;
    bool in_field = false;
    for (int i = 0; i < string->size; ++i) {
        if (isspace(string->text[i])) {
            if (in_field) {
                array_append(array, substring(string, start, i));
                in_field = false;
            }
        } else if (!in_field) {
            start = i;
            in_field = true;
        }
    }
    if (in_field) {
        array_append(array, substring(string, start, string->size));
    }
    return array;
}

String *join_strings(const Array *strings, char c) {
    assert(strings->size > 0 && "join_strings: string list is empty");
    int size = strings->size - 1;
    for (int i = 0; i < strings->size; ++i) {
        size += ((String *)strings->data[i])->size;
    }
    String *joined = malloc(sizeof(String)), *field;
    size_t block_size;
    joined->text = joined->head = malloc((size + 1) * sizeof(char));
    for (int i = 0, j = 0; i < strings->size; ++i) {
        if (i) {
            joined->text[j] = c;
            ++j;
        }
        field = strings->data[i];
        block_size = field->size * sizeof(char);
        memcpy(joined->text + j, field->text, block_size);
        j += block_size;
    }
    joined->text[size] = '\0';
    joined->size = joined->capacity = size;
    return joined;
}

bool string_start_with(const String *string, const char *prefix) {
    int size = strlen(prefix);
    if (size > string->size) {
        return false;
    }
    return !memcmp(string->text, prefix, size * sizeof(char));
}
```

A.7 core/dict_entry.h

```
#ifndef CORE_DICT_ENTRY_H_
#define CORE_DICT_ENTRY_H_

#include "core/string.h"

typedef struct DictEntry {
    String *headword, *word_class, *definition;
} DictEntry;

DictEntry *new_dict_entry(const String *line);
void delete_dict_entry(void *entry);

bool confirm(bool default_yes, const char *message);
DictEntry *input_dict_entry(const String *headword);

void display_dict_entry(const DictEntry *entry);
void write_dict_entry(const DictEntry *entry, FILE *stream);

#endif // CORE_DICT_ENTRY_H_
```

A.8 core/dict_entry.c

```
#include "core/dict_entry.h"

```

```
DictEntry *new_dict_entry(const String *line) {
    int open_index = string_index(line, '(');
    if (open_index < 0) {
        return NULL;
    }
    int close_index = -1;
    for (int i = open_index + 1, depth = 1; i < line->size; ++i) {
        if (line->text[i] == '(') {
            ++depth;
        } else if (line->text[i] == ')' && --depth <= 0) {
            close_index = i;
            break;
        }
    }
    if (close_index < 0) {
        return NULL;
    }
    DictEntry *entry = malloc(sizeof(DictEntry));
    int begin_index = 0, end_index = line->size;
    if (line->text[0] == '"' && line->text[line->size - 1] == '"') {
        ++begin_index;
        --end_index;
    }
    entry->headword = substring(line, begin_index, open_index);
    trim_in_place(entry->headword);
    if (!is_valid_key(entry->headword, false)) {
        WARN("Invalid headword: %s\n", entry->headword->text);
        delete_string(entry->headword);
        free(entry);
        return NULL;
    }
    entry->word_class = substring(line, open_index + 1, close_index);
    entry->definition = substring(line, close_index + 1, end_index);
    trim_in_place(entry->word_class);
    trim_in_place(entry->definition);
    return entry;
}

void delete_dict_entry(void *p) {
    DictEntry *entry = p;
    delete_string(entry->headword);
    delete_string(entry->word_class);
    delete_string(entry->definition);
    free(entry);
}
```

```c

bool confirm(bool default_yes, const char *message) {
    if (message) {
        printf("%s ", message);
    }
    if (default_yes) {
        printf("([y]/n) ");
    } else {
        printf("(y/[n]) ");
    }
    bool returned;
    String *line = get_line(stdin);
    trim_in_place(line);
    to_lower_in_place(line);
    if (!strcmp(line->text, "y") || !strcmp(line->text, "yes")) {
        returned = true;
    } else if (!strcmp(line->text, "n") || !strcmp(line->text, "no")) {
        returned = false;
    } else {
        returned = default_yes;
    }
    delete_string(line);
    return returned;
}

DictEntry *input_dict_entry(const String *headword) {
    if (!is_valid_key(headword, false)) {
        WARN("Invalid headword: %s\nDo nothing.\n", headword->text);
        return NULL;
    }
    DictEntry *entry = malloc(sizeof(DictEntry));
    entry->headword = trim(headword);
    printf("Word class: ");
    entry->word_class = get_line(stdin);
    printf("Definition: ");
    entry->definition = get_line(stdin);
    trim_in_place(entry->word_class);
    trim_in_place(entry->definition);
    printf("Will create the following entry:\n");
    display_dict_entry(entry);
    if (!confirm(true, "Continue?")) {
        delete_dict_entry(entry);
        printf("Do nothing.\n");
        return NULL;
    } else {
        return entry;
    }
}

void display_dict_entry(const DictEntry *entry) {
    printf("%s\n%s\n%s\n", entry->headword->text, entry->word_class->text,
           entry->definition->text);
}

void write_dict_entry(const DictEntry *entry, FILE *stream) {
    fprintf(stream, "%s (%s) %s\n", entry->headword->text,
            entry->word_class->text, entry->definition->text);
}
```

A.9 core/trie_node.h

```c
#ifndef CORE_TRIE_NODE_H_
#define CORE_TRIE_NODE_H_

#include "core/dict_entry.h"

typedef struct TrieNode {
    char letter;
    Array *entries;
    struct TrieNode *parent, *children[ALPHABET_SIZE];
    int child_count;
} TrieNode;

TrieNode *new_trie_node(char letter, TrieNode *parent);
void delete_trie_node(void *node);

TrieNode *previous_trie_node(const TrieNode *node);
TrieNode *next_trie_node(const TrieNode *node);

TrieNode *trie_node_add_child(TrieNode *node, char letter);
void trie_node_remove_child(TrieNode *node, char letter);

#endif // CORE_TRIE_NODE_H_
```

A.10 core/trie_node.c

```c
#include "core/trie_node.h"

TrieNode *new_trie_node(char letter, TrieNode *parent) {
    assert((!parent || is_valid_key_char(letter, true)) &&
           "new_trie_node: invalid character");
    TrieNode *node = malloc(sizeof(TrieNode));
    node->letter = letter;
    node->entries = new_array(delete_dict_entry);
    node->parent = parent;
    for (int i = 0; i < ALPHABET_SIZE; ++i) {
        node->children[i] = NULL;
    }
    node->child_count = 0;
    return node;
}

void delete_trie_node(void *p) {
    TrieNode *node = p;
    delete_array(node->entries);
    free(node);
}

static TrieNode *max_trie_leaf(TrieNode *node) {
    if (node->child_count <= 0) {
        return node;
    }
    for (int i = ALPHABET_SIZE - 1; i >= 0; --i) {
        if (node->children[i]) {
            return max_trie_leaf(node->children[i]);
        }
    }
    assert(false && "max_trie_leaf: unreachable code");
    return NULL;
}

TrieNode *previous_trie_node(const TrieNode *node) {
    TrieNode *parent = node->parent;
    if (!parent) {
        return NULL;
    }
    for (int i = char_to_index(node->letter) - 1; i >= 0; --i) {
        if (parent->children[i]) {
            return max_trie_leaf(parent->children[i]);
        }
    }
    return parent;
}

TrieNode *next_trie_node(const TrieNode *node) {
```

```c
    if (node->child_count > 0) {
        for (int i = 0; i < ALPHABET_SIZE; ++i) {
            if (node->children[i]) {
                return node->children[i];
            }
        }
    }
    TrieNode *parent = node->parent;
    int index;
    while (parent) {
        index = char_to_index(node->letter);
        for (int i = index + 1; i < ALPHABET_SIZE; ++i) {
            if (parent->children[i]) {
                return parent->children[i];
            }
        }
        node = parent;
        parent = node->parent;
    }
    return NULL;
}

TrieNode *trie_node_add_child(TrieNode *node, char letter) {
    int index = char_to_index(letter);
    assert(!node->children[index] &&
           "trie_node_add_child: child already exists");
    node->children[index] = new_trie_node(letter, node);
    ++node->child_count;
    return node->children[index];
}

void trie_node_remove_child(TrieNode *node, char letter) {
    int index = char_to_index(letter);
    assert(node->child_count > 0 && node->children[index] &&
           "trie_node_remove_child: child does not exist");
    assert(node->children[index]->child_count <= 0 &&
           "trie_node_remove_child: remove a non-leaf child");
    delete_trie_node(node->children[index]);
    node->children[index] = NULL;
    --node->child_count;
}
```

A.11 core/trie.h

```c
#ifndef CORE_TRIE_H_
```

```
#define CORE_TRIE_H_

#include "core/trie_node.h"

typedef struct Trie {
    TrieNode *root;
    int size;
} Trie;

Trie *new_trie();
void delete_trie(void *trie);

Array *trie_search(const Trie *trie, const String *word, bool case_sensitive);
void trie_insert(Trie *trie, DictEntry *entry);
void trie_remove(Trie *trie, const String *word, bool case_sensitive);

String *trie_predecessor(const Trie *trie, const String *word);
String *trie_successor(const Trie *trie, const String *word);

Array *traverse_trie(const Trie *trie);

#endif // CORE_TRIE_H_
```

A.12 core/trie.c

```
#include "core/trie.h"

Trie *new_trie() {
    Trie *trie = malloc(sizeof(Trie));
    trie->root = new_trie_node('\0', NULL);
    trie->size = 0;
    return trie;
}

static void clear_trie_nodes(TrieNode *root) {
    assert(root && "clear_trie_nodes: root node is null");
    if (root->child_count > 0) {
        for (int i = 0; i < ALPHABET_SIZE; ++i) {
            if (root->children[i]) {
                clear_trie_nodes(root->children[i]);
            }
        }
    }
    delete_trie_node(root);
```

```c
}

void delete_trie(void *p) {
    Trie *trie = p;
    clear_trie_nodes(trie->root);
    free(trie);
}

static TrieNode *get_trie_node(const Trie *trie, const String *word) {
    TrieNode *node = trie->root;
    for (int i = 0; i < word->size; ++i) {
        int index = char_to_index(tolower(word->text[i]));
        node = node->children[index];
        if (!node) {
            return NULL;
        }
    }
    return node;
}

Array *trie_search(const Trie *trie, const String *word, bool
   case_sensitive) {
    Array *entries = new_array(NULL);
    if (!is_valid_key(word, false)) {
        WARN("Invalid headword: %s\n", word->text);
        return entries;
    }
    TrieNode *node = get_trie_node(trie, word);
    if (!node || node->entries->size <= 0) {
        return entries;
    }
    if (case_sensitive) {
        DictEntry *entry;
        for (int i = 0; i < node->entries->size; ++i) {
            entry = node->entries->data[i];
            if (!strcmp(entry->headword->text, word->text)) {
                array_append(entries, entry);
            }
        }
    } else {
        for (int i = 0; i < node->entries->size; ++i) {
            array_append(entries, node->entries->data[i]);
        }
    }
    return entries;
```

```
}

void trie_insert(Trie *trie, DictEntry *entry) {
    assert(is_valid_key(entry->headword, false) &&
           "trie_insert: invalid entry headword");
    String *lowered = to_lower(entry->headword);
    TrieNode *node = trie->root;
    for (int i = 0; i < lowered->size; ++i) {
        int index = char_to_index(lowered->text[i]);
        if (!node->children[index]) {
            node = trie_node_add_child(node, lowered->text[i]);
        } else {
            node = node->children[index];
        }
    }
    delete_string(lowered);
    array_append(node->entries, entry);
    ++trie->size;
}

void trie_remove(Trie *trie, const String *word, bool case_sensitive) {
    assert(is_valid_key(word, false) && "trie_remove: invalid headword");
    TrieNode *node = get_trie_node(trie, word);
    if (!node) {
        return;
    }
    if (case_sensitive) {
        int i = 0;
        DictEntry *entry;
        while (i < node->entries->size) {
            entry = node->entries->data[i];
            if (!strcmp(entry->headword->text, word->text)) {
                array_remove(node->entries, i);
                --trie->size;
            } else {
                ++i;
            }
        }
    } else {
        for (int i = node->entries->size - 1; i >= 0; --i) {
            array_remove(node->entries, i);
            --trie->size;
        }
    }
    char letter;
    while (node->entries->size <= 0 && node->child_count <= 0 && node->parent) {
        letter = node->letter;
        node = node->parent;
        trie_node_remove_child(node, letter);
    }
}

String *trie_predecessor(const Trie *trie, const String *word) {
    TrieNode *node = get_trie_node(trie, word);
    if (!node) {
        WARN("Cannot find word: %s\n", word->text);
        return NULL;
    }
    node = previous_trie_node(node);
    while (node) {
        if (node->entries->size > 0) {
            DictEntry *entry = node->entries->data[0];
            return to_lower(entry->headword);
        }
        node = previous_trie_node(node);
    }
    return NULL;
}

String *trie_successor(const Trie *trie, const String *word) {
    TrieNode *node = get_trie_node(trie, word);
    if (!node) {
        WARN("Cannot find word: %s\n", word->text);
        return NULL;
    }
    node = next_trie_node(node);
    while (node) {
        if (node->entries->size > 0) {
            DictEntry *entry = node->entries->data[0];
            return to_lower(entry->headword);
        }
        node = next_trie_node(node);
    }
    return NULL;
}

static void traverse_trie_nodes(const TrieNode *root, Array *entries) {
    for (int i = 0; i < root->entries->size; ++i) {
        array_append(entries, root->entries->data[i]);
```

```c
    }
    if (root->child_count > 0) {
        for (int i = 0; i < ALPHABET_SIZE; ++i) {
            if (root->children[i]) {
                traverse_trie_nodes(root->children[i], entries);
            }
        }
    }
}

Array *traverse_trie(const Trie *trie) {
    Array *entries = new_array(NULL);
    traverse_trie_nodes(trie->root, entries);
    return entries;
}
```

A.13 core/matcher.h

```c
#ifndef CORE_TRIE_H_
#define CORE_TRIE_H_

#include "core/trie_node.h"

typedef struct Trie {
    TrieNode *root;
    int size;
} Trie;

Trie *new_trie();
void delete_trie(void *trie);

Array *trie_search(const Trie *trie, const String *word, bool
    case_sensitive);
void trie_insert(Trie *trie, DictEntry *entry);
void trie_remove(Trie *trie, const String *word, bool case_sensitive);

String *trie_predecessor(const Trie *trie, const String *word);
String *trie_successor(const Trie *trie, const String *word);

Array *traverse_trie(const Trie *trie);

#endif // CORE_TRIE_H_
```

A.14 core/matcher.c

```c
#include "core/trie.h"
```

```

Trie *new_trie() {
    Trie *trie = malloc(sizeof(Trie));
    trie->root = new_trie_node('\0', NULL);
    trie->size = 0;
    return trie;
}

static void clear_trie_nodes(TrieNode *root) {
    assert(root && "clear_trie_nodes: root node is null");
    if (root->child_count > 0) {
        for (int i = 0; i < ALPHABET_SIZE; ++i) {
            if (root->children[i]) {
                clear_trie_nodes(root->children[i]);
            }
        }
    }
    delete_trie_node(root);
}

void delete_trie(void *p) {
    Trie *trie = p;
    clear_trie_nodes(trie->root);
    free(trie);
}

static TrieNode *get_trie_node(const Trie *trie, const String *word) {
    TrieNode *node = trie->root;
    for (int i = 0; i < word->size; ++i) {
        int index = char_to_index(tolower(word->text[i]));
        node = node->children[index];
        if (!node) {
            return NULL;
        }
    }
    return node;
}

Array *trie_search(const Trie *trie, const String *word, bool
   case_sensitive) {
    Array *entries = new_array(NULL);
    if (!is_valid_key(word, false)) {
        WARN("Invalid headword: %s\n", word->text);
        return entries;
    }
```

```c
    TrieNode *node = get_trie_node(trie, word);
    if (!node || node->entries->size <= 0) {
        return entries;
    }
    if (case_sensitive) {
        DictEntry *entry;
        for (int i = 0; i < node->entries->size; ++i) {
            entry = node->entries->data[i];
            if (!strcmp(entry->headword->text, word->text)) {
                array_append(entries, entry);
            }
        }
    } else {
        for (int i = 0; i < node->entries->size; ++i) {
            array_append(entries, node->entries->data[i]);
        }
    }
    return entries;
}

void trie_insert(Trie *trie, DictEntry *entry) {
    assert(is_valid_key(entry->headword, false) &&
           "trie_insert: invalid entry headword");
    String *lowered = to_lower(entry->headword);
    TrieNode *node = trie->root;
    for (int i = 0; i < lowered->size; ++i) {
        int index = char_to_index(lowered->text[i]);
        if (!node->children[index]) {
            node = trie_node_add_child(node, lowered->text[i]);
        } else {
            node = node->children[index];
        }
    }
    delete_string(lowered);
    array_append(node->entries, entry);
    ++trie->size;
}

void trie_remove(Trie *trie, const String *word, bool case_sensitive) {
    assert(is_valid_key(word, false) && "trie_remove: invalid headword");
    TrieNode *node = get_trie_node(trie, word);
    if (!node) {
        return;
    }
    if (case_sensitive) {
        int i = 0;
        DictEntry *entry;
        while (i < node->entries->size) {
            entry = node->entries->data[i];
            if (!strcmp(entry->headword->text, word->text)) {
                array_remove(node->entries, i);
                --trie->size;
            } else {
                ++i;
            }
        }
    } else {
        for (int i = node->entries->size - 1; i >= 0; --i) {
            array_remove(node->entries, i);
            --trie->size;
        }
    }
    char letter;
    while (node->entries->size <= 0 && node->child_count <= 0 && node->parent) {
        letter = node->letter;
        node = node->parent;
        trie_node_remove_child(node, letter);
    }
}

String *trie_predecessor(const Trie *trie, const String *word) {
    TrieNode *node = get_trie_node(trie, word);
    if (!node) {
        WARN("Cannot find word: %s\n", word->text);
        return NULL;
    }
    node = previous_trie_node(node);
    while (node) {
        if (node->entries->size > 0) {
            DictEntry *entry = node->entries->data[0];
            return to_lower(entry->headword);
        }
        node = previous_trie_node(node);
    }
    return NULL;
}

String *trie_successor(const Trie *trie, const String *word) {
    TrieNode *node = get_trie_node(trie, word);
    if (!node) {
        WARN("Cannot find word: %s\n", word->text);
        return NULL;
    }
    node = next_trie_node(node);
    while (node) {
        if (node->entries->size > 0) {
            DictEntry *entry = node->entries->data[0];
            return to_lower(entry->headword);
        }
        node = next_trie_node(node);
    }
    return NULL;
}

static void traverse_trie_nodes(const TrieNode *root, Array *entries) {
    for (int i = 0; i < root->entries->size; ++i) {
        array_append(entries, root->entries->data[i]);
    }
    if (root->child_count > 0) {
        for (int i = 0; i < ALPHABET_SIZE; ++i) {
            if (root->children[i]) {
                traverse_trie_nodes(root->children[i], entries);
            }
        }
    }
}

Array *traverse_trie(const Trie *trie) {
    Array *entries = new_array(NULL);
    traverse_trie_nodes(trie->root, entries);
    return entries;
}
```

A.15 main/api.h

```
#ifndef MAIN_API_H_
#define MAIN_API_H_

#include "core/array.h"
#include "core/dict_entry.h"
#include "core/matcher.h"
#include "core/string.h"
#include "core/trie.h"
#include "core/trie_node.h"
```

```

#ifndef ESP_RC_PATH
#define ESP_RC_PATH "../.esp_rc"
#endif // ESP_RC_PATH

typedef enum EspMode {
    ESP_MODE_INTERACTIVE,
    ESP_MODE_BACKGROUND,
    ESP_MODE_COMMAND_LINE
} EspMode;

void esp_initialize(EspMode mode);
void esp_cleanup(EspMode mode);

bool esp_parse_arguments(Array *arguments, EspMode mode);

void esp_on_load(Array *arguments, EspMode mode);
void esp_on_search(Array *arguments, EspMode mode);
void esp_on_insert(Array *arguments, EspMode mode);
void esp_on_remove(Array *arguments, EspMode mode);
void esp_on_neighbour(Array *arguments, EspMode mode);
void esp_on_prefix(Array *arguments, EspMode mode);
void esp_on_match(Array *arguments, EspMode mode);
void esp_on_size(Array *arguments, EspMode mode);
void esp_on_save(Array *arguments, EspMode mode);
bool esp_on_exit(Array *arguments, EspMode mode);

#endif // MAIN_API_H_
```

A.16 main/api.c

```
#include "main/api.h"
#include "main/utility.h"

Trie *dictionary;

void esp_initialize(EspMode mode) {
    if (mode == ESP_MODE_INTERACTIVE) {
        printf("
   *=============================================\n"
               "               ||  EngSci Press Dictionary by Yunhao Qian
   ||\n"
               "
   *=============================================\n"
               "\nStarting...\n\n");
    }
    dictionary = new_trie();
    FILE *stream = fopen(ESP_RC_PATH, "r");
    if (stream) {
        String *line = get_line(stream);
        Array *arguments;
        while (line) {
            arguments = split_string(line);
            esp_parse_arguments(arguments, ESP_MODE_BACKGROUND);
            delete_array(arguments);
            delete_string(line);
            line = get_line(stream);
        }
        fclose(stream);
    }
}

void esp_cleanup(EspMode mode) {
    if (mode == ESP_MODE_INTERACTIVE) {
        printf("\nExiting...\n");
    }
    delete_trie(dictionary);
}

bool esp_parse_arguments(Array *arguments, EspMode mode) {
    if (arguments->size <= 0) {
        return true;
    }
    String *leading = to_lower(arguments->data[0]);
    array_remove(arguments, 0);
    bool returned = true;
    if (!strcmp(leading->text, "load")) {
        esp_on_load(arguments, mode);
    } else if (!strcmp(leading->text, "search")) {
        esp_on_search(arguments, mode);
    } else if (!strcmp(leading->text, "insert")) {
        esp_on_insert(arguments, mode);
    } else if (!strcmp(leading->text, "remove")) {
        esp_on_remove(arguments, mode);
    } else if (!strcmp(leading->text, "neighbour")) {
        esp_on_neighbour(arguments, mode);
    } else if (!strcmp(leading->text, "prefix")) {
        esp_on_prefix(arguments, mode);
    } else if (!strcmp(leading->text, "match")) {
        esp_on_match(arguments, mode);
    } else if (!strcmp(leading->text, "size")) {
        esp_on_size(arguments, mode);
    } else if (!strcmp(leading->text, "save")) {
        esp_on_save(arguments, mode);
    } else if (!strcmp(leading->text, "exit")) {
        returned = esp_on_exit(arguments, mode);
    } else {
        WARN("Unknown leading argument: %s\n", leading->text);
    }
    delete_string(leading);
    return returned;
}

void esp_on_load(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_COMMAND_LINE) {
        WARN_NOT_SUPPORTED("load", "command-line");
        return;
    }
    if (arguments->size < 1) {
        WARN_MISSING("file name");
        return;
    }
    if (arguments->size > 1) {
        WARN_REDUNDANT(arguments, 1);
    }
    const char *file_name = ((String *)arguments->data[0])->text;
    FILE *stream = fopen(file_name, "r");
    if (!stream) {
        WARN("Cannot open file: %s\nDo nothing.\n", file_name);
        return;
    }
    String *line = get_line(stream);
    DictEntry *entry;
    int count = 0;
    while (line) {
        if (line->size > 0) {
            entry = new_dict_entry(line);
            if (!entry) {
                WARN("Failed to parse the following line in %s:\n%s\n",
                     file_name, line->text);
            } else {
                trie_insert(dictionary, entry);
                ++count;
            }
        }
        delete_string(line);
        line = get_line(stream);
    }
    fclose(stream);
    if (mode == ESP_MODE_INTERACTIVE) {
        printf("%d entries loaded from %s\n", count, file_name);
    }
}

void esp_on_search(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_BACKGROUND) {
        WARN_NOT_SUPPORTED("search", "background");
        return;
    }
    if (arguments->size <= 0) {
        WARN_MISSING("headword");
        return;
    }
    String *word = join_strings(arguments, ' ');
    bool case_sensitive = false;
    for (int i = 0; i < word->size; ++i) {
        if (isupper(word->text[i])) {
            case_sensitive = true;
            break;
        }
    }
    Array *results = trie_search(dictionary, word, case_sensitive);
    if (results->size <= 0) {
        WARN("Find no entry named: %s\n", word->text);
        word_hint(word, dictionary, case_sensitive);
    } else {
        for (int i = 0; i < results->size; ++i) {
            putchar('\n');
            display_dict_entry(results->data[i]);
        }
        putchar('\n');
    }
    delete_array(results);
    delete_string(word);
}

void esp_on_insert(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_BACKGROUND) {
        WARN_NOT_SUPPORTED("insert", "background");
        return;
    }
    if (mode == ESP_MODE_COMMAND_LINE) {
        WARN_NOT_SUPPORTED("insert", "command-line");
        return;
    }
    if (arguments->size <= 0) {
        WARN_MISSING("headword");
        return;
    }
    String *headword = join_strings(arguments, ' ');
    DictEntry *entry = input_dict_entry(headword);
    if (entry) {
        trie_insert(dictionary, entry);
    }
    delete_string(headword);
}

void esp_on_remove(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_COMMAND_LINE) {
        WARN_NOT_SUPPORTED("remove", "command-line");
        return;
    }
    if (arguments->size <= 0) {
        WARN_MISSING("headword");
        return;
    }
    String *headword = join_strings(arguments, ' ');
    bool case_sensitive = false;
    for (int i = 0; i < headword->size; ++i) {
        if (isupper(headword->text[i])) {
            case_sensitive = true;
            break;
        }
    }
    Array *results = trie_search(dictionary, headword, case_sensitive);
    int remove_count = results->size;
    bool shall_remove = true;
    if (remove_count <= 0) {
        WARN("Find no entry named: %s\n", headword->text);
        word_hint(headword, dictionary, case_sensitive);
        shall_remove = false;
    } else if (mode == ESP_MODE_INTERACTIVE) {
        printf("The following entries will be removed:\n");
        for (int i = 0; i < remove_count; ++i) {
            putchar('\n');
```

```c
            display_dict_entry(results->data[i]);
        }
        shall_remove = confirm(true, "\nWant to continue?");
    }
    delete_array(results);
    if (shall_remove) {
        trie_remove(dictionary, headword, case_sensitive);
        if (mode == ESP_MODE_INTERACTIVE) {
            printf("%d entries removed.\n", remove_count);
        }
    } else if (mode == ESP_MODE_INTERACTIVE) {
        printf("Do nothing.\n");
    }
}

void esp_on_neighbour(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_BACKGROUND) {
        WARN_NOT_SUPPORTED("neighbour", "background");
        return;
    }
    if (arguments->size <= 0) {
        WARN_MISSING("headword");
        return;
    }
    int radius = 10;
    if (string_start_with(arguments->data[0], "--")) {
        if (arguments->size <= 1) {
            WARN_MISSING("headword");
            return;
        }
        int number = parse_unsigned_int_flag(arguments->data[0]);
        if (number < 0) {
            WARN("Invalid flag: %s\n", ((String *)arguments->data[0])->text
   );
        } else {
            radius = number;
        }
        array_remove(arguments, 0);
    }
    String *word = join_strings(arguments, ' ');
    delete_string(word);
}

void esp_on_prefix(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_BACKGROUND) {
        WARN_NOT_SUPPORTED("prefix", "background");
        return;
    }
    if (arguments->size <= 0) {
        WARN_MISSING("prefix string");
        return;
    }
    int max_count = 10;
    if (string_start_with(arguments->data[0], "--")) {
        if (arguments->size <= 1) {
            WARN_MISSING("prefix string");
            return;
        }
        int number = parse_unsigned_int_flag(arguments->data[0]);
        if (number < 0) {
            WARN("Invalid flag: %s\n", ((String *)arguments->data[0])->text);
        } else {
            max_count = number;
        }
        array_remove(arguments, 0);
    }
    String *prefix = join_strings(arguments, ' ');
    delete_string(prefix);
}

void esp_on_match(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_BACKGROUND) {
        WARN_NOT_SUPPORTED("match", "background");
        return;
    }
    if (arguments->size <= 0) {
        WARN_MISSING("headword");
        return;
    }
    int tolerance = -1;
    if (string_start_with(arguments->data[0], "--")) {
        if (arguments->size <= 1) {
            WARN_MISSING("headword");
            return;
        }
        int number = parse_unsigned_int_flag(arguments->data[0]);
        if (number < 0) {
            WARN("Invalid flag: %s\n", ((String *)arguments->data[0])->text);
```

```c
        } else {
            tolerance = number;
        }
        array_remove(arguments, 0);
    }
    String *pattern = join_strings(arguments, ' ');
    String *matched = trie_closest_match(dictionary, pattern, tolerance);
    if (!matched) {
        WARN("Find no entry similar to: %s\n", pattern->text);
    } else {
        printf("%s\n", matched->text);
        delete_string(matched);
    }
    delete_string(pattern);
}

void esp_on_size(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_BACKGROUND) {
        WARN_NOT_SUPPORTED("size", "background");
        return;
    }
    if (arguments->size > 0) {
        WARN_REDUNDANT(arguments, 0);
    }
    printf("Dictionary size: %d\n", dictionary->size);
}

void esp_on_save(Array *arguments, EspMode mode) {
    if (arguments->size < 1) {
        WARN_MISSING("file name");
        return;
    }
    if (arguments->size > 1) {
        WARN_REDUNDANT(arguments, 1);
    }
    if (mode == ESP_MODE_INTERACTIVE && dictionary->size <= 0 &&
        !confirm(false, "The dictionary is empty. Continue?")) {
        printf("Do nothing.\n");
        return;
    }
    const char *file_name = ((String *)arguments->data[0])->text;
    FILE *stream = fopen(file_name, "wb");
    if (!stream) {
        WARN("Cannot open file: %s\nDo nothing.\n", file_name);
        return;
    }
    Array *entries = traverse_trie(dictionary);
    for (int i = 0; i < entries->size; ++i) {
        write_dict_entry(entries->data[i], stream);
    }
    delete_array(entries);
    fclose(stream);
    if (mode == ESP_MODE_INTERACTIVE) {
        printf("%d entries saved to %s.\n", dictionary->size, file_name);
    }
}

bool esp_on_exit(Array *arguments, EspMode mode) {
    if (mode == ESP_MODE_BACKGROUND) {
        WARN_NOT_SUPPORTED("exit", "background");
        return true;
    }
    if (mode == ESP_MODE_COMMAND_LINE) {
        WARN_NOT_SUPPORTED("exit", "command-line");
        return true;
    }
    if (arguments->size > 0) {
        WARN_REDUNDANT(arguments, 0);
    }
    return !confirm(true, "Are you sure you want to exit?");
}
```

A.17 main/utility.h

```c
#ifndef MAIN_UTILITY_H_
#define MAIN_UTILITY_H_

#include "main/api.h"

#define WARN_NOT_SUPPORTED(argument, mode)
       \
    WARN("Does not support \"%s\" in %s mode.\n", argument, mode);

#define WARN_MISSING(expected)
       \
    WARN("Missing argument: %s expected.\n", expected)

#define WARN_REDUNDANT(arguments, start_index)
       \
    WARN("Redundant arguments: ignore arguments since \"%s\".\n",
       \
         ((String *)(arguments)->data[start_index])->text)

int parse_unsigned_int_flag(const String *string);

void word_hint(const String *string, const Trie *dictionary,
               bool case_sensitive);

#endif // MAIN_UTILITY_H_
```

A.18 main/utility.c

```c
#include "main/utility.h"

int parse_unsigned_int_flag(const String *string) {
    assert(string_start_with(string, "--") &&
           "parse_unsigned_int_flag: not a flag");
    if (string->size <= 2) {
        return -1;
    }
    String *flag = substring(string, 2, string->size);
    for (int i = 0; i < flag->size; ++i) {
        if (!isdigit(flag->text[i])) {
            delete_string(flag);
            return -1;
        }
    }
    int number = atoi(flag->text);
    delete_string(flag);
    return number;
}

void word_hint(const String *string, const Trie *dictionary,
               bool case_sensitive) {
    String *matched = trie_closest_match(dictionary, string, -1);
    if (!matched) {
        return;
    }
    if (case_sensitive) {
        String *lowered = to_lower(string);
        if (!strcmp(matched->text, lowered->text)) {
            printf("Tip: use lower-case word for case-insensitive "
                   "search/remove.\n");
            delete_string(lowered);
            return;
        }
        delete_string(lowered);
    }
    printf("Did you mean: %s\n", matched->text);
    delete_string(matched);
}
```

A.19 main/main.c

```
#include "main/api.h"

int main(int argc, const char **argv) {
    Array *arguments;
    if (argc > 1) {
        arguments = new_array(delete_string);
        for (int i = 1; i < argc; ++i) {
            array_append(arguments, new_string(argv[i], -1));
        }
        esp_initialize(ESP_MODE_COMMAND_LINE);
        esp_parse_arguments(arguments, ESP_MODE_COMMAND_LINE);
        delete_array(arguments);
        esp_cleanup(ESP_MODE_COMMAND_LINE);
        return 0;
    }
    esp_initialize(ESP_MODE_INTERACTIVE);
    String *line;
    bool shall_continue;
    do {
        printf(">>> ");
        line = get_line(stdin);
        if (!line) {
            break;
        }
        arguments = split_string(line);
        shall_continue = esp_parse_arguments(arguments, ESP_MODE_INTERACTIVE);
        delete_array(arguments);
        delete_string(line);
    } while (shall_continue);
    esp_cleanup(ESP_MODE_INTERACTIVE);
}
```

A.20 writer/grammar.py

```
from random import choices, random
```

```python
from re import compile, match


float_pattern = compile(r'(([0-9]*\.)?[0-9]+)\s*\:\s+')
token_pattern = compile(
    r'(([^\s\"\(\)]|(\(([^\(\)]|(\"([^\"]|\\")+\"))+\))|(\"([^\"]|\\")*\"))
   +)\s*')


class Token:

    __slots__ = 'identifier', 'terminal', 'optional', 'probability'

    def __init__(self, string):
        self.terminal = False
        self.optional = False
        self.probability = 1
        if string.startswith('(') and string.endswith(')'):
            string = string[1:-1]
            matched = float_pattern.match(string)
            factor = 1
            if matched:
                string = string[matched.end():]
                factor = float(matched.group(1))
            self.__init__(string)
            self.optional = True
            self.probability *= factor
        elif string.startswith('"') and string.endswith('"'):
            self.identifier = string[1:-1].replace('\\"', '"')
            self.terminal = True
        else:
            self.identifier = string

    def __eq__(self, other):
        if not isinstance(other, Token):
            return False
        return self.identifier == other.identifier and \
            self.terminal == other.terminal

    def __hash__(self):
        return hash((self.identifier, self.terminal))

    def __str__(self):
        string = self.identifier.replace('"', '\\"')
        if self.terminal:
            string = '"{}"'.format(string)
        if self.optional:
            if self.probability == 1:
                string = '({})'.format(string)
            else:
                string = '({}: {})'.format(self.probability, string)
        return string


class Rule:

    __slots__ = 'lhs', 'rhs', 'weight'

    def __init__(self, string):
        matched = float_pattern.match(string)
        if matched:
            string = string[matched.end():]
            self.weight = float(matched.group(1))
        else:
            self.weight = 1
        matched = token_pattern.match(string)
        string = string[matched.end():]
        self.lhs = Token(matched.group(1))
        string = string[match(r'->\s*', string).end():]
        self.rhs = []
        while True:
            matched = token_pattern.match(string)
            if not matched:
                break
            string = string[matched.end():]
            self.rhs.append(Token(matched.group(1)))

    def __eq__(self, other):
        if not isinstance(other, Rule):
            return False
        return self.lhs, tuple(self.rhs), self.weight == \
            other.lhs, tuple(other.rhs), other.weight

    def __hash__(self):
        return hash((self.lhs, tuple(self.rhs), self.weight))

    def __str__(self):
        elements = []
        if self.weight != 1:
            elements.append(str(self.weight) + ':')
        elements += [str(self.lhs), '->']
        for element in self.rhs:
            elements.append(str(element))
        return ' '.join(elements)


class CFG:

    __slots__ = 'rules', 'convergence'

    def __init__(self, string=None):
        self.rules = {}
        self.convergence = 1
        if string:
            self.load_lines(string)

    def __str__(self):
        elements = []
        if self.convergence != 1:
            elements.append('convergence = {}'.format(self.convergence))
        for rule_list in self.rules.values():
            for rule in rule_list:
                elements.append(str(rule))
        return '\n'.join(elements)

    def load_line(self, line):
        line = line.strip()
        if line == '' or line.startswith('//'):
            return
        matched = match(r'convergence\s*=\s*(([0-9]*\.)?[0-9]+)\s*', line)
        if matched:
            self.convergence = float(matched.group(1))
            return
        rule = Rule(line)
        if rule.lhs in self.rules:
            self.rules[rule.lhs].append(rule)
        else:
            self.rules[rule.lhs] = [rule]

    def load_lines(self, string):
        for line in string.split('\n'):
            try:
                self.load_line(line)
            except:
                print('Failed to parse line: {}'.format(line))

    def generate(self, start=Token('S'), max_length=-1):
        weight_dict = {}
        for rule_list in self.rules.values():
            for rule in rule_list:
                weight_dict[rule] = rule.weight
        stack = [start]
        terminals = []
        while len(stack) > 0:
            token = stack.pop()
            if token.optional and random() > token.probability:
                continue
            if token.terminal:
                terminals.append(token.identifier)
                continue
            try:
                rule_list = self.rules[token]
            except:
                raise Exception('Failed to find rule for: {}'.format(token)
    )
            weights = [weight_dict[rule] for rule in rule_list]
            rule = choices(rule_list, weights, k=1)[0]
            weight_dict[rule] *= self.convergence
            stack += rule.rhs[::-1]
            if max_length > 0 and len(terminals) > max_length:
                raise Exception('Exceed max length: {}'.format(max_length))
        return terminals


demo_grammar = '''
convergence = 0.3

0.9: S -> Clause "."
0.1: S -> Clause "while" S

Clause -> NP VP

0.9: NP -> Det (0.6: Adj) N
0.1: NP -> NP "and" NP

VP -> V NP
VP -> V

Det -> "a"
Det -> "the"
```

```

Adj -> "smart"
Adj -> "tired"
Adj -> "brown"

N -> "student"
N -> "laptop"
N -> "car"

V -> "drives"
V -> "walks"
V -> "leaves"
'''


if __name__ == '__main__':
    cfg = CFG(demo_grammar)
    print(cfg)
    while True:
        input('------------------------------------------------------------')
        try:
            tokens = cfg.generate(max_length=30)
        except Exception as exception:
            print(exception)
        else:
            tokens[0] = tokens[0].capitalize()
            print(' '.join(tokens[:-1]) + tokens[-1])
```

A.21 writer/lexicon.py

```
from collections import defaultdict


class DictEntry:

    __slots__ = 'headword', 'word_class'

    def __init__(self, headword, word_class):
        self.headword = headword
        self.word_class = word_class

    @staticmethod
    def from_line(line):
        if line.startswith('"') and line.endswith('"'):
            line = line[1:-1]
        left_index = line.index('(')
        if left_index < 0:
            raise Exception('missing word class', line)
        right_index = -1
        depth = 1
        for i in range(left_index + 1, len(line)):
            if line[i] == '(':
                depth += 1
            elif line[i] == ')':
                depth -= 1
                if depth == 0:
                    right_index = i
                    break
        if right_index < 0:
            raise Exception('mismatched brackets', line)
        headword = line[:left_index].strip()
        if headword == '':
            raise Exception('empty headword', line)
        word_class = line[left_index + 1:right_index].strip()
        return DictEntry(headword, word_class)


pos_tag_to_word_classes = {
    'Proper-Noun-Sg': set(),
    'Proper-Noun-Pl': set(),
    'Noun-Sg': {
        'n. & v',
        'n.& v.',
        'n & v.',
        'n. & v. t.',
        'n.',
        'n. sing & pl.',
        'a & n.',
        'n. & v. i.',
        'n. /',
        'n. / interj.',
        'n. & v.',
        'sing. or pl.',
        'n.sing & pl.',
        'n',
        'n., a., & v.',
        'n. & a.',
        'sing. & pl.',
        'n .',
        'v. t. & n.',
        'n. sing. & pl.',
        'a., n., & adv.',
        'n. & adv.',
        'n. / v. t. & i.',
        'n.sing. & pl.',
        'n. .',
        'v.& n.',
        'n. & interj.',
        'adv. & n.',
        'n. Chem.',
        'v. i. & n.',
        'n.',
        'sing.',
        'N.',
        'n./',
        'adv., & n.',
        'a. / n.',
        'v. & n.',
        'a., adv., & n.',
        'n..',
        'n. sing. & pl',
        'interj. & n.',
        'n. sing.',
        'n. & i.',
        'imperative sing.',
        'syntactically sing.'
    },
    'Noun-Pl': {
        'n. pl.',
        'n. sing & pl.',
        'n.pl.',
        'sing. or pl.',
        'n.sing & pl.',
        'sing. & pl.',
        'n. pl',
        'n. sing. & pl.',
        'n.sing. & pl.',
        'n pl.',
        'n., sing. & pl.',
        'n. collect. & pl.',
        'n. sing. & pl',
        'n. pl.',
        'sing. / pl.'
    },
    'Gerund-V': {
        'p. pr. & v. n.',
        'p. pr. &, vb. n.',
        'imp. & p. p. Fenced (/); p. pr. & vb. n.',
        'imp. & p. p. & vb. n.',
        'p, pr. & vb. n.',
        'p. pr. a. & vb. n.',
        'p. pr. vb. n.',
        'imp. & p. pr. & vb. n.',
        'pr.p. & vb. n.',
        'p. pr. / vb. n.',
        'p]. pr. & vb. n.',
        'p. pr.& vb. n.',
        'p. pr. &vb. n.',
        'p. pr. & vb/ n.',
        'P. pr. & vb. n.',
        'p. pr. & vvb. n.',
        'p. a. & vb. n.',
        'p. pr. &. vb. n.',
        'p. pr. & pr. & vb. n.',
        'vb. n.',
        'p. p. & vb. n.',
        'p pr. & vb. n.',
        'imp. & p. p. Adored (/); p. pr. & vb. n.',
        'p. pr & vb. n.'
    },
    'Verb-I-Sg': {
        '3d sing.pr.',
        'subj. 3d pers. sing.',
        '3d sing.',
        '3d pers. sing. pres.',
        '3d sing. pr.',
        'pres. indic. sing., 1st & 3d pers.',
        'Sing. pres. ind.',
        '3d sing.',
        'pres. sing.'
    },
    'Verb-T-Sg': {
        '3d sing.pr.',
        'subj. 3d pers. sing.',
        '3d sing.',
        '3d pers. sing. pres.',
        '3d sing. pr.',
        'pres. indic. sing., 1st & 3d pers.',
        'Sing. pres. ind.',
        '3d sing.',
        'pres. sing.'
    },
    'Verb-I-Pl': {
        'v. t. / i.',
        'v. i.,',
        'n. & v. i.',
        'v. i. & i.',
        'v. i.',
        'v.t & i.',
        'v.i',
        'v. t. / v. i.',
        'v.i.',
        'v. t.& i.',
        'n. / v. t. & i.',
        'v. i.',
        'v. t. & v. i.',
        'v. i. & n.',
        'v. i. & auxiliary.',
        'v. t. & i.',
        'v. i. & t.',
        'v. i. / auxiliary'
    },
    'Verb-T-Pl': {
        'v. t. / i.',
        'a. & v. t.',
        'v. t. &',
        'n. & v. t.',
        'v. t..',
        'v. t. v. t.',
        'v.t & i.',
        'v. t. / v. i.',
        'v.t',
        'v. t. & n.',
        'v. t.& i.',
        'n. / v. t. & i.',
        'v. t. & v. i.',
        'v./t.',
        'v. t.',
        'v. t. / auxiliary',
        'v. t.',
        'v. i. & t.',
        'v.t.'
    },
    'Preposition': {
        'prep., adv., & conj.',
        'prep., adv., conj. & n.',
        'adv. & prep.',
        'prep. & conj., but properly a participle',
        'prep., adv. & a.',
        'prep., adv. & conj.',
        'prep. & adv.',
        'adv., prep., & conj.',
        'prep.',
        'adv. or prep.',
        'prep. & conj.',
        'conj. & prep.'
    },
    'Adj': {
        'adj.',
        'pron. / adj.',
        'a.',
        'p. p. / a.',
        'a. & v. t.',
        'adv. & a.',
        'p. p & a.',
        'p. p. & a.',
        'a. / a. pron.',
        'P. p. & a.',
        'pron. & a.',
        'a & n.',
        'a/',
        'adv. / a.',
        'a. & a. pron.',
        'a & p. p.',
        'p. & a.',
        'prep., adv. & a.',
        'a. .',
        'a. superl.',
        'v. & a.',
        'a. & adv.',
        'n., a., & v.',
        'a. a.',
        'pron., a., conj., & adv.',
        'n. & a.',
        'p. pr. a. & vb. n.',
        'a. & v.',
        'a., n., & adv.',
        'a. Vigorously',
        'a. & n.',
        'a.',
        'a. / adv.',
        'a & adv.',
        'a. Vibrating',
        'a. or pron.',
        'a. / pron.',
        'imp., p. p., & a.',
        'a',
        'p. p. & a',
        'a. / n.',
        'pron., a., & adv.',
        'a., adv., & n.',
        'a. & p. p.',
        'a. & pron.'
    },
    'Adv': {
        'prep., adv., & conj.',
        'prep., adv., conj. & n.',
        'adv. & a.',
        'adv. In combination or cooperation',
        'adv. / interj.',
        'interrog. adv.',
        'adv. & prep.',
        'adv. In a vanishing manner',
        'adv. / a.',
        'prep., adv. & a.',
        'a. & adv.',
        'pron., a., conj., & adv.',
        'prep., adv. & conj.',
        'conj. / adv.',
        'adv.',
        'prep. & adv.',
        'interj., adv., or a.',
        'a., n., & adv.',
        'interj., adv., & n.',
        'n. & adv.',
        'a. / adv.',
        'adv., prep., & conj.',
        'adv. & n.',
        'a & adv.',
        'adv. or prep.',
        'adv., & n.',
        'pron., a., & adv.',
        'a., adv., & n.',
        'interj. & adv.',
        'adv. / conj.',
        'adv. & conj.'
    }
}


def create_lexicon(lines):
    lexicon = {}
    for tag in pos_tag_to_word_classes:
        lexicon[tag] = set()
    for line in lines:
        line = line.strip()
        if line == '':
            continue
        try:
            entry = DictEntry.from_line(line)
        except:
            print('Failed to parse line: {}'.format(line))
            continue
        for tag in pos_tag_to_word_classes:
            word_class_set = pos_tag_to_word_classes[tag]
            if entry.word_class in word_class_set:
                lexicon[tag].add(entry.headword.lower().replace('"', '\\"')
   )
    return lexicon


def write_lexicon(lexicon, stream):
    for tag in pos_tag_to_word_classes:
        for terminal in lexicon[tag]:
            stream.write('{} -> "{}"\n'.format(tag, terminal))


if __name__ == '__main__':
    try:
        output = open('espg_lexicon.txt', 'w')
    except:
        print('Failed to open file: {}'.format('espg_lexicon.txt'))
        exit()
    for i in range(ord('A'), ord('Z') + 1):
        file_name = ('../../Dictionary-in-csv/{}.csv'.format(chr(i)))
        try:
            with open(file_name, 'r') as stream:
                lines = open(file_name, 'r').readlines()
        except:
            print('Failed to open file: {}'.format(file_name))
        else:
            lexicon = create_lexicon(lines)
            write_lexicon(lexicon, output)
    output.close()
```

A.22 writer/espg_base.txt

```
// ===== Sentence =====
S -> NP-Sg VP-Sg "."
S -> NP-Pl VP-Pl "."
0.4: S -> VP-Pl "!"
0.2: S -> Aux-Sg NP-Sg VP-Pl "?"
0.2: S -> Aux-Pl NP-Pl VP-Pl "?"
0.1: S -> Wh-NP-Sg Aux-Sg NP-Sg VP-Pl "?"
0.1: S -> Wh-NP-Pl Aux-Pl NP-Pl VP-Pl "?"

// ===== Noun Phrase =====
0.2: NP-Sg -> Pronoun-Sg
// NP-Sg -> Proper-Noun-Sg
NP-Sg -> Det-Sg (0.5: AP) Nominal-Sg
0.2: NP-Pl -> Pronoun-Pl
// NP-Pl -> Proper-Noun-Pl
NP-Pl -> Det-Pl (0.5: AP) Nominal-Pl

// ===== Nominal =====
Nominal-Sg -> Noun-Sg
0.3: Nominal-Sg -> Nominal-Sg PP
0.3: Nominal-Sg -> Nominal-Sg Gerund-VP
0.3: Nominal-Sg -> Nominal-Sg Rel-Clause-Sg
Nominal-Pl -> Noun-Pl
0.3: Nominal-Pl -> Nominal-Pl PP
0.3: Nominal-Pl -> Nominal-Pl Gerund-VP
0.3: Nominal-Pl -> Nominal-Pl Rel-Clause-Pl

// ===== Gerundive Verb =====
Gerund-VP -> Gerund-V
Gerund-VP -> Gerund-V NP-Sg
Gerund-VP -> Gerund-V NP-Pl
Gerund-VP -> Gerund-V PP
Gerund-VP -> Gerund-V NP-Sg PP
Gerund-VP -> Gerund-V NP-Pl PP

// ===== Relative Clause =====
Rel-Clause-Sg -> Rel-Pronoun VP-Sg
```

```
Rel-Clause-Pl -> Rel-Pronoun VP-Pl

// ===== Verb Phrase =====
VP-Sg -> Verb-I-Sg
VP-Sg -> Verb-T-Sg NP-Sg
VP-Sg -> Verb-T-Sg NP-Pl
VP-Sg -> Verb-T-Sg NP-Sg PP
VP-Sg -> Verb-T-Sg NP-Pl PP
VP-Sg -> Verb-I-Sg PP
VP-Pl -> Verb-I-Pl
VP-Pl -> Verb-T-Pl NP-Sg
VP-Pl -> Verb-T-Pl NP-Pl
VP-Pl -> Verb-T-Pl NP-Sg PP
VP-Pl -> Verb-T-Pl NP-Pl PP
VP-Pl -> Verb-I-Pl PP

// ===== Adjective Phrase =====
AP -> Adj
0.2: AP -> Adv AP

// ===== Prepositional Phrase =====
PP -> Preposition NP-Sg
PP -> Preposition NP-Pl

// ===== Determiner =====
5: Det-Sg -> "the"
5: Det-Pl -> "the"
5: Det-Sg -> "a"
4: Deg-Sg -> "this"
4: Deg-Sg -> "that"
4: Det-Pl -> "these"
4: Det-Pl -> "those"
Det-Sg -> "my"
Det-Pl -> "my"
Det-Sg -> "your"
Det-Pl -> "your"
Det-Sg -> "his"
Det-Pl -> "his"
Det-Sg -> "her"
Deg-Pl -> "her"
Det-Sg -> "its"
Det-Pl -> "its"
Det-Sg -> "our"
Det-Pl -> "our"
Det-Sg -> "their"
```

```
Det-Pl -> "their"
2: Det-Pl -> "a" "few"
2: Det-Pl -> "many"
2: Det-Pl -> "a" "lot" "of"
3: Det-Pl -> "some"
Det-Sg -> "any"
Det-Sg -> "one"
Det-Pl -> "all"
Det-Sg -> "each"
Det-Sg -> "every"
Det-Sg -> "another"
Det-Sg -> NP-Sg "'s"
Det-Pl -> NP-Sg "'s"

// ===== Auxiliary Verb =====
Aux-Sg -> "has"
Aux-Pl -> "have"
Aux-Sg -> "had"
Aux-Pl -> "had"
Aux-Sg -> "did"
Aux-Pl -> "did"
Aux-Sg -> "will"
Aux-Pl -> "will"
Aux-Sg -> "should"
Aux-Pl -> "should"
Aux-Sg -> "would"
Aux-Pl -> "would"
Aux-Sg -> "may"
Aux-Pl -> "may"
Aux-Sg -> "might"
Aux-Pl -> "might"
Aux-Sg -> "must"
Aux-Pl -> "must"
Aux-Sg -> "can"
Aux-Pl -> "can"
Aux-Sg -> "could"
Aux-Pl -> "could"
Aux-Sg -> "does"
Aux-Pl -> "do"
Aux-Sg -> "need"
Aux-Pl -> "need"

// ===== Wh- Noun Phrase ====
Wh-NP-Sg -> "when"
Wh-NP-Pl -> "when"
```

```
Wh-NP-Sg -> "who"
Wh-NP-Pl -> "who"
Wh-NP-Sg -> "where"
Wh-NP-Pl -> "where"
Wh-NP-Sg -> "what"
Wh-NP-Pl -> "what"
Wh-NP-Sg -> "what" Noun-Sg
Wh-NP-Pl -> "what" Noun-Pl
Wh-NP-Sg -> "whose" Noun-Sg
Wh-NP-Pl -> "whose" Noun-Pl
Wh-NP-Sg -> "which" Noun-Sg
Wh-NP-Pl -> "which" Noun-Pl

// ===== Pronoun =====
4: Pronoun-Pl -> "you"
Pronoun-Sg -> "yours"
Pronoun-Pl -> "yours"
2: Pronoun-Pl -> "yourself"
Pronoun-Sg -> "him"
Pronoun-Sg -> "his"
Pronoun-Pl -> "his"
2: Pronoun-Sg -> "himself"
Pronoun-Sg -> "her"
Pronoun-Sg -> "hers"
Pronoun-Pl -> "hers"
2: Pronoun-Sg -> "herself"
4: Pronoun-Sg -> "it"
Pronoun-Sg -> "its"
Pronoun-Pl -> "its"
Pronoun-Sg -> "itself"
4: Pronoun-Sg -> "ours"
4: Pronoun-Pl -> "ours"
2: Pronoun-Pl -> "ourself"
Pronoun-Sg -> "theirs"
Pronoun-Pl -> "theirs"
2: Pronoun-Pl -> "themselves"

// ===== Relative Pronoun =====
Rel-Pronoun -> "who"
Rel-Pronoun -> "which"
Rel-Pronoun -> "that"

// ===== In espg_lexicon.txt =====
// Noun-Sg
// Noun-Pl
// Gerund-V
// Verb-I-Sg
// Verb-T-Sg
// Verb-I-Pl
// Verb-T-Pl
// Preposition
// Adj
// Adv
```

A.23 writer/writer.py

```python
from string import punctuation
from numpy.random import poisson


class Sentence:

    __slots__ = 'tokens'

    def __init__(self, tokens):
        self.tokens = tokens

    def __str__(self):
        elements = []
        for token in self.tokens:
            if len(elements) > 0:
                if token[0] in punctuation or elements[-1][-1] in punctuation:
                    elements[-1] += token
                else:
                    elements.append(token)
            else:
                elements.append(token.capitalize())
        return ' '.join(elements)

    def word_count(self):
        count = 0
        for token in self.tokens:
            if token not in punctuation:
                count += 1
        return count


class Paragraph:

    __slots__ = 'sentences', 'indent'

    def __init__(self, indent):
        self.sentences = []
        self.indent = indent

    def __str__(self):
        elements = []
        for sentence in self.sentences:
            elements.append(str(sentence))
        return ' ' * self.indent + ' '.join(elements)

    def add_sentence(self, tokens):
        self.sentences.append(Sentence(tokens))

    def word_count(self):
        count = 0
        for sentence in self.sentences:
            count += sentence.word_count()
        return count


class Article:

    __slots__ = 'paragraphs', 'title', 'spacing', 'indent'

    def __init__(self, title, spacing, indent):
        self.paragraphs = []
        self.title = title
        self.spacing = spacing
        self.indent = indent

    def __str__(self):
        elements = []
        if self.title:
            elements.append(self.title)
        for paragraph in self.paragraphs:
            if len(paragraph.sentences) == 0:
                continue
            elements.append(str(paragraph))
        return ('\n' * (self.spacing + 1)).join(elements)

    def add_sentence(self, tokens):
        if len(self.paragraphs) == 0:
            self.new_paragraph()
        self.paragraphs[-1].add_sentence(tokens)

    def new_paragraph(self, indent=None):
        if len(self.paragraphs) > 0 and \
                len(self.paragraphs[-1].sentences) == 0:
            self.paragraphs.pop()
        if indent == None:
            indent = self.indent
        self.paragraphs.append(Paragraph(indent))

    def word_count(self):
        count = 0
        for paragraph in self.paragraphs:
            count += paragraph.word_count()
        return count


class Writer:

    __slots__ = 'grammar', 'paragraphs_per_article',\
        'sentences_per_paragraph', 'tokens_per_sentence'

    def __init__(self, grammar, paragraphs_per_article=5,
                 sentences_per_paragraph=10, tokens_per_sentence=20):
        self.grammar = grammar
        self.paragraphs_per_article = paragraphs_per_article
        self.sentences_per_paragraph = sentences_per_paragraph
        self.tokens_per_sentence = tokens_per_sentence

    def generate(self, title=None, spacing=1, indent=4):
        article = Article(title, spacing, indent)
        for i in range(poisson(self.paragraphs_per_article)):
            for j in range(poisson(self.sentences_per_paragraph)):
                attempt_count = 0
                while True:
                    try:
                        max_length = poisson(self.tokens_per_sentence)
                        tokens = self.grammar.generate(max_length=max_length)
                    except:
                        attempt_count += 1
                        if attempt_count > 50:
                            raise Exception('Too many attempts')
                    else:
                        article.add_sentence(tokens)
                        break
            article.new_paragraph()
        return article


if __name__ == '__main__':
    from grammar import CFG, demo_grammar
    article = Writer(CFG(demo_grammar)).generate('Demo Article')
    print(article)
    print()
    print('Word count: {}'.format(article.word_count()))
```

A.24 writer/main.py

```
from grammar import CFG
from writer import Writer


def create_espg():
    espg = CFG()
    with open('espg_base.txt', 'r') as stream:
        lines = stream.read()
    espg.load_lines(lines)
    with open('espg_lexicon.txt', 'r') as stream:
        lines = stream.read()
    espg.load_lines(lines)
    return espg


if __name__ == '__main__':
    espg = create_espg()
    writer = Writer(espg, 3, 10, 15)
    while True:
        input('------------------------------------------------------------')
        article = writer.generate()
        print(article)
        print()
        print('Word count: {}'.format(article.word_count()))
```