```
/**
 * \file spell_check.c
 *
 * Skeleton code for a spell checking program.
 *
 * \author eaburns
 * \date 04-08-2010
 */

#define _GNU_SOURCE             /* for strnlen() from string.h */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <ctype.h>

#include <sys/time.h>

#if !defined(LINE_MAX)
#if !defined(_POSIX2_LINE_MAX)
#define LINE_MAX 4096           /* should be large enough. */
#else
#define LINE_MAX _POSIX2_LINE_MAX
#endif  /* !_POSIX2_LINE_MAX */
#endif  /* !LINE_MAX */

static int add_delete = 0;
static int next_word(FILE *infile, char w[], unsigned int n);
static const char* tstring = "trie";
static const char* lstring = "list";

typedef unsigned int (*spell_check)(char *[], unsigned int, FILE*,
                                    FILE*, unsigned int);

static unsigned int check_one_scan(char *dict[], unsigned int n, char *word,
                                   FILE *outfile, unsigned int edits);


struct trie_node * init_trie(char *dict[], unsigned int n);

unsigned int char_index(char c);

char * in_trie(char * word);

static unsigned int check_trie_scan(struct trie_node * root, char *word,
                                   FILE *outfile, unsigned int edits);

struct trie_node * trie_root;


/*
 * A binary tree node for storing suggested spelling corrections.
 * This allows duplicates to be removed and as an additional benefit
 * the correctins can be output in alphabetical order.
 *
 * A trie may be better for this, but of course we don’t want to
 * include the solution in the assignment.
 */
struct sugg_node {
        char *word;
        struct sugg_node *left;
        struct sugg_node *right;
};

/*
 * Creates a new suggestion tree node which is returned via the ’np’
 * argument.
```

```
 *
 * Return 0 on success and 1 on error.
 */
static unsigned int create_sugg_node(struct sugg_node **np, char *word)
{
        struct sugg_node *n;

        n = malloc(sizeof(*n));
        if (!n) {
                perror("malloc failed");
                return 1;
        }
        n->word = word;
        n->left = NULL;
        n->right = NULL;

        *np = n;

        return 0;
}


/*
 * Adds a new suggestion into the tree and return the new tree root
 * via the ’root’ argument.  If the same word is already in the tree,
 * it is not added twice.
 *
 * Return 0 on success and 1 on error.
 */
static unsigned int add_sugg(struct sugg_node **root, char *word)
{
        unsigned int err = 0;
        int cmp;

        if (!*root) {
                err = create_sugg_node(root, word);
                if (err)
                        return 1;
                return 0;
        }

        cmp = strcmp(word, (*root)->word);
        if (cmp < 0)
                err = add_sugg(&(*root)->left, word);
        else if (cmp > 0)
                err = add_sugg(&(*root)->right, word);

        return err;
}


/* Outputs the suggestions to the given file. */
static void output_suggs(FILE *outfile, struct sugg_node *root)
{
        if (root) {
                output_suggs(outfile, root->left);
                fprintf(outfile, "\t%s\n", root->word);
                output_suggs(outfile, root->right);
        }
}


/* Frees the suggestion tree. */
static void free_suggs(struct sugg_node *root)
{
        if (root) {
                free_suggs(root->left);
                free_suggs(root->right);
                free(root);
```

```c
        }
}

/** List method */
static unsigned int check_edits(char **dict, unsigned int n, char *word,
                                FILE *outfile, unsigned int edits,
                                struct sugg_node **suggs);


static unsigned int check_trie_edits(char *word,
                                FILE *outfile, unsigned int edits,
                                struct sugg_node **suggs);

static unsigned int check_trie_subs(struct trie_node * root, char *word,
                               char * current_word, FILE *outfile, int edits, in
t depth,
                               struct sugg_node **suggs);

static unsigned int trie_edit(char *word,
                             FILE *outfile, unsigned int edits,
                             struct sugg_node **suggs);


/************************************************************
 * You sholud change the trie_spell_check() function to insert the
 * dictionary words into a trie.  Then, instead of calling the
 * check_one_scan() function, you sholud perform all lookups in the
 * trie.
 *
 * Pay careful attention to the format of the output from the
 * check_one_scan() function and make sure that your final program has
 * matching output.
 ************************************************************/

/************************************************************
 * Trie data structures
 ************************************************************/

typedef struct trie_node trie_node;

struct trie_node{
        char * word;
        struct trie_node * references[52];
};


/*
 * Insert all words from the dictionary into a trie.
 * Return 0 on success and 1 on error.
 */
static unsigned int trie_spell_check(char *dict[], unsigned int n, FILE *wfile,
                                FILE *outfile, unsigned int edits){
    int ret;
    unsigned int err;
    char word[LINE_MAX + 1];

    trie_root = init_trie(dict, n);

    ret = next_word(wfile, word, LINE_MAX + 1);
    while(ret == 0){
                err = check_trie_scan(trie_root, word, outfile, edits);
                if (err) return 1;
        ret = next_word(wfile, word, LINE_MAX + 1);
    }
        return 0;
}
```

```c
/**
 * Allocate a trie_node and return a pointer to it.
 */
struct trie_node * make_trie_node(){
        int i = 0;

        struct trie_node * node = malloc(sizeof(struct trie_node));
        for(i = 0; i < 52; i++){
                node->references[i] = malloc(sizeof(struct trie_node));
        }
        return node;
}

/**
 * Convert a character to an index into the references[52] array.
 * Lowercase characters are on the bottom half of the array, upper case
 * characters start at 26.
 */
unsigned int char_index(char c){
        if(islower(c)){
                return c − 97;
        }
        return c − 39;
}

/**
 * Convert a character to an index into the references[52] array.
 * Lowercase characters are on the bottom half of the array, upper case
 * characters start at 26.
 */
char index_char(int c){
        if(c < 26){
                return c + 97;
        }
        return c + 39;
}



/**
 * Initialize the trie by adding everything in dict.
 */
struct trie_node * init_trie(char *dict[], unsigned int n){
        int i, j, len, index = 0;
        char * word;
        struct trie_node * root = make_trie_node();
        struct trie_node * cur = root;

        for(i = 0; i < n; i++){
                word = dict[i];
                len = strlen(word);
                for(j = 0; j < len; j++){
                        index = char_index(word[j]);
                        if(cur−>references[index] == NULL){
                                cur−>references[index] = make_trie_node();
                        }
                        cur = cur−>references[index];
                }
                /* we are at the end of the word */
                cur−>word = strdup(word);
                /*fprintf(stderr, "added %s to trie\n", cur−>word);*/
                cur = root;
        }
        return root;
}

/**
 * Returns the word if the word was found in the trie,
 * or NULL if it is not in the trie.
 */
char * in_trie(char * word)
{
        unsigned int i, index, len = 0;
        struct trie_node * cur = trie_root;

        len = strlen(word);
        for(i = 0; i < len; i++){
                index = char_index(word[i]);
                if(cur->references[index] == NULL){
                        return NULL;
                }
                cur = cur->references[index];
        }
        return cur -> word;
}


/**
 * If the word is found in the trie, return.
 * If it is not, check for suggestions with an edit distance of 'edits'.
 */
static unsigned int check_trie_scan(struct trie_node * root, char *word,
                                   FILE *outfile, unsigned int edits)
{
        unsigned int err;
        struct sugg_node *suggs = NULL;

#if !defined(NDEBUG)
        fprintf(stderr, "Checking [%s]\n", word);
#endif  /* !NDEBUG */

        if (in_trie(word)) {
                fprintf(outfile, "correct: %s\n", word);
        } else {
                fprintf(outfile, "incorrect: %s\n", word);
                err = check_trie_edits(word, outfile, edits, &suggs);
                if (err) {
                        free_suggs(suggs);
                        return 1;
                }
                fprintf(outfile, "suggestions:\n");
                output_suggs(outfile, suggs);
                fprintf(outfile, "\t----\n");
                free_suggs(suggs);
        }
        return 0;
}


/*
 * Check if any add/delete/substitute edits are in the trie with an
 * edit distance of 'edits'.
 * c a t s
 */
static unsigned int check_trie_edits(char *word,
                                FILE *outfile, unsigned int edits,
                                struct sugg_node **suggs){

        int word_len;
        char * cur_word;
        unsigned int err;


        if (edits > 0) {
                /* check substitutions */
                word_len = strlen(word);
                cur_word = calloc(word_len+1, sizeof(char));
                err = check_trie_subs(trie_root, word, cur_word, outfile, edits,
```

```c
 0, suggs);
                if (err)
                        return 1;
                /*if (add_delete == 1){
                        err = check_adds(root word, outfile, edits, suggs);
                        if (err)
                                return 1;
                }
                if (add_delete == 1){
                        err = check_dels(root, word, outfile, edits, suggs);
                        if (err)
                                return 1;
                }*/
        }
        return 0;
}

/**
 * Check for substitution edits in the trie.
 * cat -> bat counts as a subtitution edit of '1'.
 * Word is the rest of the word that must be matched.
 * If an edit is found, pass it to trie_edit which will
 * recurse on that edit.
 */
static unsigned int check_trie_subs(struct trie_node * root, char *word,
                               char * cur_word, FILE *outfile, int edits, int de
pth,
                               struct sugg_node **suggs)
{
        int i, len = 0;
        struct trie_node * cur = root;
        char * word_dup;

        len = strlen(word);

        /* only do two edits */
        if(edits < 0){
                return 0;
        }

        /* don't check words larger than one we are looking for */
        if(depth > len)
                return 0;

        if(depth == len){
                if(cur->word){
                        trie_edit(cur_word, outfile, edits, suggs);
                }
        }

        /* update the current word */
        word_dup = calloc(len+1, sizeof(char));
        strcpy(word_dup, cur_word);

        for(i = 0; i < 52; i++){
                if(cur->references[i]){
                        cur_word[depth] = index_char(i);
                        cur_word[depth+1] = '\0';
                        if(cur_word[depth] == word[depth]){
                                check_trie_subs(cur->references[i], word, cur_wo
rd, outfile, edits, depth+1, suggs);
                        }
                        else{
                                check_trie_subs(cur->references[i], word, cur_wo
rd, outfile, edits-1, depth+1, suggs);
                        }
                }
        }
        return 0;
```

```
}


/*
 * Check the dictionary for the given edit.  Recurs to try more edits
 * of this edit too.
 *
 * Return 0 on success and 1 on error.
 */
static unsigned int trie_edit(char *word,
                              FILE *outfile, unsigned int edits,
                              struct sugg_node **suggs)
{
        unsigned int err;
        char *found;

        /*fprintf(stderr, "trying %s\n", word);*/

        found = in_trie(word);
        /*fprintf(stderr, "found: %s\n", found);*/
        if (found) {
#if !defined(NDEBUG)
                fprintf(stderr, "Adding suggestion [%s]\n", word);
#endif  /* !NDEBUG */
                err = add_sugg(suggs, found);
                if (err)
                        return 1;
        }
        err = check_trie_edits(word, outfile, edits - 1, suggs);
        if (err)
                return 1;

        return 0;
}



/*
 * Read in the words and check them against the dictionary.
 * Return 0 on success and 1 on error.
 */
static unsigned int list_spell_check(char *dict[], unsigned int n, FILE *wfile,
                                FILE *outfile, unsigned int edits){
        int ret;
        unsigned int err;
        char word[LINE_MAX + 1];

        ret = next_word(wfile, word, LINE_MAX + 1);
        while (ret == 0) {
                err = check_one_scan(dict, n, word, outfile, edits);
                if (err) return 1;
                ret = next_word(wfile, word, LINE_MAX + 1);
        }

        return 0;
}


/************************************************************
 * Some I/O routines.
 ************************************************************/

/* Eat characters until a space is found. */
static void eat_till_space(FILE *infile)
{
        while (!isspace(getc(infile)))
                ;
```

```
}


/* Eats whitespace and returns the first non-whitespace character (or
 * EOF). */
static int eat_space(FILE *infile)
{
        int c;

        do {
                c = getc(infile);
        } while (isspace(c));

        return c;
}


/* Adds 'c' to the 'i'th index of 'w' (performs bounds checking on the
 * array). */
static void add_to_word(FILE *infile, int c, unsigned int i,
                        char w[], unsigned int n)
{
        if (i >= n) {
                fprintf(stderr, "Word is too long: truncating");
                eat_till_space(infile);
                w[n - 1] = '\0';
        } else {
                w[i] = c;
        }
}


/*
 * Reads the next word from the given input file.  The word is stored in the 'w'
 buffer which must have at least 'n' characters available.
 *
 * Return 0 on success or EOF if the end of file was reached (in which
 * case 'w' is left in an unknown state.
 */
static int next_word(FILE *infile, char w[], unsigned int n)
{
        int c;
        unsigned int i = 0;

        c = eat_space(infile);
        while (c != EOF) {
                if (isalpha(c)) {
                        add_to_word(infile, c, i, w, n);
                        i += 1;
                } else if (isspace(c)) {
                        add_to_word(infile, '\0', i, w, n);
                        break;
                } else {
                        fprintf(stderr, "Non-alpha '%c', skipping word\n", c);
                        eat_till_space(infile);
                        return next_word(infile, w, n);
                }
                c = getc(infile);
        }

        if (c == EOF)
                return EOF;

        assert(w[i] == '\0');
        {
                unsigned int j;
                for (j = 0; j < i; j += 1)
                        assert(isalpha(w[j]));
                assert(strlen(w) == i);
```

```c
        }

        return 0;
}


/*
 * Reads the next word from the input file into an exact-fit string
 * and returns it via the 'word' argument.  The caller is responsable
 * for freeing the return value.
 *
 * The return value is 0 on success, 1 on error or EOF if the end of
 * file was reached (in which case 'word' is left unchanged).
 */
static int read_exact_fit_word(FILE *infile, char **word)
{
        int ret;
        size_t len;
        char word_buf[LINE_MAX + 1];

        ret = next_word(infile, word_buf, LINE_MAX + 1);
        if (ret == EOF)
                return EOF;

        len = strnlen(word_buf, LINE_MAX + 1);
        *word = malloc(sizeof(**word) * (len + 1));
        if (!*word) {
                perror("malloc failed");
                return 1;
        }

        assert(word_buf[len] == '\0');

        strncpy(*word, word_buf, sizeof(**word) * (len + 1));

        assert((*word)[len] == '\0');
        assert(strlen(*word) == len);

        return 0;
}


/************************************************************
 * Reading words into an array.
 ************************************************************/

/*
 * Grows the array from 'size' to 'new_size' and returns the new array
 * or NULL on error.
 */
static char **grow_words_ary(char *ary[], unsigned int size,
                             unsigned int new_size)
{
        unsigned int i;

        ary = realloc(ary, sizeof(*ary) * new_size);
        if (!ary) {
                perror("realloc failed");
                return NULL;
        }

        for (i = size; i < new_size; i += 1)
                ary[i] = NULL;

        return ary;
}


/*
```

```c
 * Frees the memory allocated for the words.
 */
static void free_words(char *words[], unsigned int n)
{
        if (words) {
                unsigned int i;
                for (i = 0; i < n; i += 1) {
                        if (words[i])
                                free(words[i]);
                }
                free(words);
        }
}


/*
 * Reads the words from the input file.  The return value is an array
 * of words or NULL on error.  The number of words that were read is
 * returned through the argument 'n'.
 */
static char **read_words(FILE *infile, unsigned int *n)
{
        unsigned int nwords = 0;
        unsigned int nalloced = 100;
        int ret;
        char **words;

        words = grow_words_ary(NULL, 0, nalloced);
        if (!words)
                return NULL;

        ret = read_exact_fit_word(infile, &words[nwords]);
        nwords += 1;
        while(ret == 0) {
                ret = read_exact_fit_word(infile, &words[nwords]);
                nwords += 1;
                if (nwords == nalloced) {
                        words = grow_words_ary(words, nalloced,
                                               nalloced * 2);
                        if (!words)
                                return NULL;
                        nalloced *= 2;
                }
        }

        if (ret != EOF)
                free_words(words, nwords);

        *n = nwords - 1;

        return words;
}


/************************************************************
 * Dealing with suggested spellings.
 ************************************************************/



/************************************************************
 * An example function that performs the checks by a linear scan.
 ************************************************************/

/*
 * Scan the dictionary and check for an occurance of 'word'.  If the
 * word is found then a pointer to the word in the dictionary is
 * returned.  If the word is not found then NULL is returned.
 */
```

```c
static char *in_dict(char *dict[], unsigned int n, char *word)
{
	unsigned int i;

	for (i = 0; i < n; i += 1) {
		if (strcmp(dict[i], word) == 0)
			return dict[i];
	}

	return NULL;
}



/*
 * Check the dictionary for the given edit.  Recurs to try more edits
 * of this edit too.
 *
 * Return 0 on success and 1 on error.
 */
static unsigned int try_edit(char *dict[], unsigned int n, char *word,
			     FILE *outfile, unsigned int edits,
			     struct sugg_node **suggs)
{
	unsigned int err;
	char *found;

	found = in_dict(dict, n, word);
	if (found) {
#if !defined(NDEBUG)
		fprintf(stderr, "Adding suggestion [%s]\n", word);
#endif	/* !NDEBUG */
		err = add_sugg(suggs, found);
		if (err)
			return 1;
	}
	err = check_edits(dict, n, word, outfile, edits − 1, suggs);
	if (err)
		return 1;

	return 0;
}


/*
 * Check if any substitution edits are in the dictionary.
 *
 * Returns 0 on success and 1 on error.
 */
static unsigned int check_subs(char *dict[], unsigned int n, char *word,
			       FILE *outfile, unsigned int edits,
			       struct sugg_node **suggs)
{
	int i;
	size_t len;
	unsigned int err;

	len = strlen(word);
	for (i = 0; i < len; i += 1) {
		char s;
		char c = word[i];

		for (s = ’A’; s <= ’Z’; s += 1) {
			if (s != c) {
				word[i] = s;
				err = try_edit(dict, n, word, outfile,
					       edits, suggs);
				if (err)
					return 1;
```

```c
				}
			}
			for (s = ’a’; s <= ’z’; s += 1) {
				if (s != c) {
					word[i] = s;
					err = try_edit(dict, n, word, outfile,
						       edits, suggs);
					if (err)
						return 1;
				}
			}

			word[i] = c;
		}

		return 0;
}


/*
 * Copies the word from ’src’ into ’dst’ with a gap at a given index
 * ’gindex’.  ’len’ is the length of the source buffer.
 */
static void copy_with_gap(char *dst, unsigned int gindex, char *src,
			  unsigned int len)
{
	unsigned int i, j;

	for (i = j = 0; i < len; i += 1, j += 1) {
		if (i == gindex)
			j += 1;
		dst[j] = src[i];
	}
	dst[j] = ’\0’;
}

/*
 * Check if any adds are in the dictionary.
 *
 * Returns 0 on success and 1 on failure.
 */
static unsigned int check_adds(char *dict[], unsigned int n, char *word,
			       FILE *outfile, unsigned int edits,
			       struct sugg_node **suggs)
{
	char *word2;
	int i;
	size_t len;
	unsigned int err;

	len = strlen(word);
	word2 = malloc(sizeof(*word2) * (len + 2));
	if (!word2) {
		perror("malloc failed");
		return 1;
	}

	for (i = 0; i <= len; i += 1) {
		char c;

		copy_with_gap(word2, i, word, len);

		for (c = ’A’; c <= ’Z’; c += 1) {
			word2[i] = c;
			assert(strlen(word2) == len + 1);
			err = try_edit(dict, n, word2, outfile, edits, suggs);
			if (err)
				return 1;
		}
		for (c = 'a'; c <= 'z'; c += 1) {
			word2[i] = c;
			assert(strlen(word2) == len + 1);
			err = try_edit(dict, n, word2, outfile, edits, suggs);
			if (err)
				return 1;
		}
	}

	free(word2);
	return 0;
}


/*
 * Copies 'src' into 'dst' except the character at 'dindex' is left
 * off.
 */
static void copy_with_del(char *dst, unsigned int dindex, char *src,
			  unsigned int len)
{
	unsigned int i, j;

	for (i = j = 0; i < len; i += 1) {
		if (i != dindex) {
			dst[j] = src[i];
			j += 1;
		}
	}

	dst[j] = '\0';
}


/*
 * Check if any deletes are in the dictionary.
 *
 * Returns 0 on success and 1 on failure.
 */
static unsigned int check_dels(char *dict[], unsigned int n, char *word,
			       FILE *outfile, unsigned int edits,
			       struct sugg_node **suggs)
{
	char *word2;
	int i;
	size_t len;
	unsigned int err;

	len = strlen(word);
	word2 = malloc(sizeof(*word2) * len);
	if (!word2) {
		perror("malloc failed");
		return 1;
	}

	for (i = 0; i < len; i += 1) {
		copy_with_del(word2, i, word, len);
		assert(strlen(word2) == len - 1);
		err = try_edit(dict, n, word2, outfile, edits, suggs);
		if (err)
			return 1;
	}

	free(word2);
	return 0;
}


/*
```

```c
 * Check if any add/delete/substitute edits are in the dictionary.
 */
static unsigned int check_edits(char *dict[], unsigned int n, char *word,
				FILE *outfile, unsigned int edits,
				struct sugg_node **suggs){
	if (edits > 0) {
		unsigned int err;

		err = check_subs(dict, n, word, outfile, edits, suggs);
		if (err)
			return 1;
		if (add_delete == 1){
			err = check_adds(dict, n, word, outfile, edits, suggs);
			if (err)
				return 1;
		}
		if (add_delete == 1){
			err = check_dels(dict, n, word, outfile, edits, suggs);
			if (err)
				return 1;
		}
	}

	return 0;
}


/*
 * Check one word against the dictionary using a scan of the entire
 * dictionary.
 *
 * Return 0 on success and 1 on error.
 */
static unsigned int check_one_scan(char *dict[], unsigned int n, char *word,
				   FILE *outfile, unsigned int edits)
{
#if !defined(NDEBUG)
	fprintf(stderr, "Checking [%s]\n", word);
#endif  /* !NDEBUG */

	if (in_dict(dict, n, word)) {
		fprintf(outfile, "correct: %s\n", word);
	} else {
		unsigned int err;
		struct sugg_node *suggs = NULL;

		fprintf(outfile, "incorrect: %s\n", word);
		err = check_edits(dict, n, word, outfile, edits, &suggs);
		if (err) {
			free_suggs(suggs);
			return 1;
		}
		fprintf(outfile, "suggestions:\n");
		output_suggs(outfile, suggs);
		fprintf(outfile, "\t----\n");
		free_suggs(suggs);
	}

	return 0;
}


/* Gets the time of day in seconds. */
static double get_current_seconds(void)
{
    double sec, usec;
    struct timeval tv;

    if (gettimeofday(&tv, NULL) < 0) {
        perror("gettimeofday failed");
        exit(EXIT_FAILURE);
    }

    sec = tv.tv_sec;
    usec = tv.tv_usec;

    return sec + (usec / 1000000);
}


/* Read the dictionary, check the words. */
static unsigned int read_dict_and_check_words(FILE *dfile,
                                              FILE *wfile,
                                              FILE *outfile,
                                              unsigned int edits,
                                              spell_check s)
{
        char **dict;
        unsigned int n = 0;
        unsigned int err;
        double start_time, end_time;

        dict = read_words(dfile, &n);
        if (!dict)
                return 1;

        start_time = get_current_seconds();
        err = s(dict, n, wfile, outfile, edits);
        if (err) {
                free_words(dict, n);
                return 1;
        }
        end_time = get_current_seconds();

        fprintf(outfile, "time: %f seconds\n", end_time - start_time);

        free_words(dict, n);

        return 0;
}



/*************************************************************
 * The main function.
 *************************************************************/


/* Print the words to the given file. */
/*
static void print_words(FILE *out, char **words, unsigned int n)
{
        unsigned int i;

        for (i = 0; i < n; i += 1)
                fprintf(out, "%s\n", words[i]);
}
*/


/* print the usage message and then exit with failure status. */
static void usage(void)
{
    printf("Usage:\nspell_check [--adds-deletes] <alg> <dictionary> <words> <outfile>\nwhere alg one of {list
,trie}\n");
    exit(EXIT_FAILURE);
}
```

```c

int main (int argc, char const *argv[])
{
    unsigned int err;
    int ret = EXIT_SUCCESS;
    FILE *dict;
    FILE *words = stdin;
    FILE *outfile = stdout;
    int i = 0, j = 0;

    if (argc > 6 || argc < 5)
        usage();

    if (argc == 6){
        if (strcmp(argv[1], "--adds-deletes") == 0){
            add_delete = i = j = 1;
        }
        else if (strcmp(argv[2], "--adds-deletes") == 0){
            add_delete = i = 1;
        }
        else if (strcmp(argv[5], "--adds-deletes") == 0){
            add_delete = 1;
        }
        else {
            usage();
        }
    }

    dict = fopen(argv[2+i], "r");
    if (!dict) {
        perror("Error opening dictionary");
        goto out;
    }

    if (strcmp(argv[3+i], "-") != 0) {
        words = fopen(argv[3+i], "r");
        if (!words) {
            perror("Error opening words file");
            goto out;
        }
    }

    if (strcmp(argv[4+i], "-") != 0) {
        outfile = fopen(argv[4+i], "w");
        if (!outfile) {
            perror("Error opening output file");
            goto out;
        }
    }

    if(strcmp(argv[1+j],tstring) == 0)
        err = read_dict_and_check_words(dict, words, outfile, 2,
                        &trie_spell_check);
    else if (strcmp(argv[1+j],lstring) == 0)
        err = read_dict_and_check_words(dict, words, outfile, 2,
                        &list_spell_check);
    else{
        printf("I don't know what to do with that alg!\n"
               "I got %s, but I expected %s or %s\n", argv[1+j],
               tstring, lstring);
        err = 1;
    }
    if (err)
        ret = EXIT_FAILURE;

out:
    if (dict)
        fclose(dict);
    if (words && words != stdin)
        fclose(words);

    if (outfile && outfile != stdout)
        fclose(outfile);

    return ret;
}
```