Part 1 – Shell Source Code

https://github.com/NootCode/CS2600-Final/tree/master/Part1-Shell

#include <sys/wait.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

char \*lsh\_read\_line(void){

    #ifdef LSH\_USE\_STD\_GETLINE

    char \*line = NULL;

    ssize\_t bufsize = 0;

    if(getline(&line, &bufsize, stdin) == -1){

        if(feof(stdin)){

            exit(EXIT\_SUCCESS);

        }else{

            perror("readline");

            exit(EXIT\_FAILURE);

        }

    }

    return line;

#else

#define LSH\_RL\_BUFSIZE 1024

    int bufsize = LSH\_RL\_BUFSIZE;

    int position = 0;

    char \*buffer = malloc(sizeof(char)\*bufsize);

    int c;

    if(!buffer){

        fprintf(stderr, "lsh: allocation error\n");

        exit(EXIT\_FAILURE);

    }

    while(1){

        c = getchar();

        if(c == EOF || c == '\n'){

            buffer[position] = '\0';

            return buffer;

        } else{

            buffer[position] = c;

        }

        position++;

        if(position >= bufsize){

            bufsize += LSH\_RL\_BUFSIZE;

            buffer = realloc(buffer, bufsize);

            if(!buffer){

                fprintf(stderr, "lsh: allocation error\n");

                exit(EXIT\_FAILURE);

            }

        }

    }

#endif

}

#define LSH\_TOK\_BUFSIZE 64

#define LSH\_TOK\_DELIM " \t\r\n\a"

char \*\*lsh\_split\_line(char \*line){

    int bufsize = LSH\_TOK\_BUFSIZE, position = 0;

    char \*\*tokens = malloc(bufsize \* sizeof(char\*));

    char \*token;

    if(!tokens){

        fprintf(stderr, "lsh: allocation error\n");

        exit(EXIT\_FAILURE);

    }

    token = strtok(line, LSH\_TOK\_DELIM);

    while(token != NULL){

        tokens[position] = token;

        position++;

        if(position >= bufsize){

            bufsize += LSH\_TOK\_BUFSIZE;

            tokens = realloc(tokens, bufsize \* sizeof(char\*));

             if(!tokens){

                fprintf(stderr, "lsh: allocation error\n");

                exit(EXIT\_FAILURE);

            }

        }

        token = strtok(NULL, LSH\_TOK\_DELIM);

    }

    tokens[position] = NULL;

    return tokens;

}

int lsh\_launch(char \*\*args){

    pid\_t pid, wpid;

    int status;

    pid = fork();

    if(pid == 0){

        if(execvp(args[0], args) == -1){

            perror("lsh");

        }

        exit(EXIT\_FAILURE);

    }else if (pid > 0){

        perror("lsh");

    }else{

        do{

            wpid = waitpid(pid, &status, WUNTRACED);

        }while(!WIFEXITED(status) && !WIFSIGNALED(status));

    }

    return 1;

}

int lsh\_cd(char \*\*args);

int lsh\_help(char \*\*args);

int lsh\_exit(char \*\*args);

char \*builtin\_str[] = {

    "cd", "help", "exit"

};

int(\*builtin\_func[])(char\*\*)= {

    &lsh\_cd,

    &lsh\_help,

    &lsh\_exit

};

int lsh\_num\_builtins(){

    return sizeof(builtin\_str) / sizeof(char \*);

}

int lsh\_cd(char \*\*args){

    if(args[1] == NULL){

        fprintf(stderr, "lsh: expected argument to \"cd\"\n");

    }else{

        if(chdir(args[1]) != 0){

            perror("lsh");

        }

    }

    return 1;

}

int lsh\_help(char \*\*args){

    int i;

    printf("Andre Nalbandians LSH\n");

    printf("Type Program names and Arguments and hit enter.\n");

    printf("The following are built in: \n");

    for(i = 0; i < lsh\_num\_builtins(); i++){

        printf(" %s\n", builtin\_str[i]);

    }

    printf("Use the man command for info on other programs.\n");

    return 1;

}

int lsh\_exit(char \*\*args){

    return 0;

}

int lsh\_execute(char \*\*args){

    int i;

    if(args[0] == NULL){

        return 1;

    }

    for(i = 0; i < lsh\_num\_builtins(); i++){

        if(strcmp(args[0], builtin\_str[i]) == 0){

            return (\*builtin\_func[i])(args);

        }

    }

    return lsh\_launch(args);

}

void lsh\_loop(void){

    char \*line;

    char \*\*args;

    int status;

    do{

        printf("> ");

        line = lsh\_read\_line();

        args = lsh\_split\_line(line);

        status = lsh\_execute(args);

        free(line);

        free(args);

    }while(status);

}

int main(int argc, char \*\*argv){

    lsh\_loop();

    return EXIT\_SUCCESS;

}

Part 2 – Text Editor Source Code

<https://github.com/NootCode/CS2600-Final/blob/master/Part2-Text/kilo.c>

/\*\* includes \*\*/

#define \_DEFAULT\_SOURCE

#define \_BSD\_SOURCE

#define \_GNU\_SOURCE

#include <unistd.h>

#include <termios.h>

#include <stdlib.h>

#include <fcntl.h>

#include <ctype.h>

#include <stdio.h>

#include <errno.h>

#include <sys/ioctl.h>

#include <stdarg.h>

#include <string.h>

#include <time.h>

#include <sys/types.h>

/\*\* defines \*\*/

# define CTRL\_KEY(k) ((k) & 0x1f)

# define KILO\_VERSION "0.0.1"

# define KILO\_TAB\_STOP 8

# define KILO\_QUIT\_TIMES 3

enum editorKey{

    BACKSPACE = 127,

    ARROW\_LEFT = 1000,

    ARROW\_RIGHT,

    ARROW\_UP,

    ARROW\_DOWN,

    HOME\_KEY,

    END\_KEY,

    DEL\_KEY,

    PAGE\_UP,

    PAGE\_DOWN

};

enum editorHighlight {

    HL\_NORMAL = 0,

    HL\_COMMENT,

    HL\_MLCOMMENT,

    HL\_KEYWORD1,

    HL\_KEYWORD2,

    HL\_STRING,

    HL\_NUMBER,

    HL\_MATCH,

};

#define  HL\_HIGHLIGHT\_NUMBERS (1<<0)

#define  HL\_HIGHLIGHT\_STRINGS (1<<1)

/\*\* data \*\*/

struct editorSyntax{

    char \*filetype;

    char \*\*filematch;

    char \*\* keywords;

    char \*singleline\_comment\_start;

    char \*multiline\_comment\_start;

    char \*multiline\_comment\_end;

    int flags;

};

typedef struct erow{

    int idx;

    int size;

    int rsize;

    char \*chars;

    char \*render;

    unsigned char \*hl;

    int hl\_open\_comment;

}erow;

struct editorConfig{

    int cx, cy;

    int rx;

    int rowoff;

    int coloff;

    int screenrows;

    int screencols;

    int numrows;

    erow \*row;

    int dirty;

    char \*filename;

    char statusmsg[80];

    time\_t statusmsg\_time;

    struct editorSyntax \*syntax;

    struct termios orig\_termios;

};

struct editorConfig E;

/\*\* filetypes \*\*/

char \*C\_HL\_extensions[] = { ".c", ".h", ".cpp", NULL};

char \*C\_HL\_keywords[] = {

    "switch", "if", "while", "for", "break", "continue", "return", "else",

    "struct", "union", "typedef", "static", "enum", "class", "case",

    "int|", "long|", "double|", "float|", "char|", "unsigned|", "signed|",

    "void|", NULL

};

struct editorSyntax HLDB[] = {

    {

        "c",

        C\_HL\_extensions,

        C\_HL\_keywords,

        "//", "/\*", "\*/",

        HL\_HIGHLIGHT\_NUMBERS | HL\_HIGHLIGHT\_STRINGS

    },

};

#define HLDB\_ENTRIES (sizeof(HLDB) / sizeof(HLDB[0]))

/\*\* prototypes \*\*/

void editorSetStatusMessage(const char \*fmt, ...);

void editorRefreshScreen();

char \*editorPrompt(char \*prompt, void (\*callback)(char \*, int));

/\*\* terminal \*\*/

void die(const char \*s){

    write(STDOUT\_FILENO, "\x1b[2J", 4);

    write(STDOUT\_FILENO, "\x1b[H",3);

    perror(s);

    exit(1);

}

void disableRawMode(){

    if(tcsetattr(STDIN\_FILENO, TCSAFLUSH, &E.orig\_termios) == -1)

        die("tcsetattr");

}

void enableRawMode(){

    if(tcgetattr(STDIN\_FILENO, &E.orig\_termios) == -1)

        die("tcgetattr");

    atexit(disableRawMode);

    struct termios raw = E.orig\_termios;

    raw.c\_iflag &= ~(BRKINT| ICRNL| INPCK| ISTRIP | IXON);

    raw.c\_oflag &= ~(OPOST);

    raw.c\_cflag |= (CS8);

    raw.c\_lflag &= ~(ECHO | ICANON | IEXTEN | ISIG);

    raw.c\_cc[VMIN] = 0;

    raw.c\_cc[VTIME] = 1;

    tcsetattr(STDIN\_FILENO, TCSAFLUSH, &raw);

}

int editorReadKey(){

    int nread;

    char c;

    while((nread = read(STDIN\_FILENO, &c, 1)) != 1){

        if(nread == 01 && errno != EAGAIN)

            die("read");

    }

    if(c == '\x1b'){

        char seq[3];

        if(read(STDIN\_FILENO, &seq[0], 1) != 1) return '\x1b';

        if(read(STDIN\_FILENO, &seq[1], 1) != 1) return '\x1b';

        if(seq[0] == '['){

            if(seq[1] >= '0' && seq[1] <= '9'){

                if(read(STDIN\_FILENO, &seq[2], 1) != 1) return '\x1b';

                if(seq[2] =='~'){

                    switch(seq[1]){

                        case '1': return HOME\_KEY;

                        case '3': return DEL\_KEY;

                        case '4': return END\_KEY;

                        case '5': return PAGE\_UP;

                        case '6': return PAGE\_DOWN;

                        case '7': return HOME\_KEY;

                        case '8': return END\_KEY;

                    }

                }

            }else{

                switch (seq[1]){

                    case 'A': return ARROW\_UP;

                    case 'B': return ARROW\_DOWN;

                    case 'C': return ARROW\_RIGHT;

                    case 'D': return ARROW\_LEFT;

                    case 'H': return HOME\_KEY;

                    case 'F': return END\_KEY;

                }

            }

        } else if (seq[0] == 'O'){

            switch(seq[1]){

                case 'H': return HOME\_KEY;

                case 'F': return END\_KEY;

            }

        }

        return '\x1b';

    }else{

        return c;

    }

}

int getCursorPosition(int \*rows, int \*cols){

    char buf[32];

    unsigned int i = 0;

    if(write(STDOUT\_FILENO, "\x1b[6n" ,4) != 4) return -1;

    while(i < sizeof(buf) -1 ){

        if(read(STDIN\_FILENO, &buf[i], 1) != 1) break;

        if(buf[i] == 'R') break;

        i++;

    }

    buf[i] = '\0';

    if(buf[0] != '\x1b' || buf[1] != '[') return -1;

    if(sscanf(&buf[2], "%d;%d", rows, cols) != 2)return -1;

    return 0;

}

int getWindowSize(int \*rows, int \*cols){

    struct winsize ws;

    if(ioctl(STDOUT\_FILENO, TIOCGWINSZ, &ws) == 01 || ws.ws\_col == 0){

        if(write(STDOUT\_FILENO, "\x1b[999C\x1b[999B", 12) != 12)

            return -1;

        return getCursorPosition(rows,cols);

    }else{

        \*cols = ws.ws\_col;

        \*rows = ws.ws\_row;

        return 0;

    }

}

/\*\* syntax highlighting \*\*/

int is\_separator(int c){

    return isspace(c) || c == '\0' || strchr(",.()+-/\*=~%<>[];", c) != NULL;

}

//done

void editorUpdateSyntax(erow \*row) {

  row->hl = realloc(row->hl, row->rsize);

  memset(row->hl, HL\_NORMAL, row->rsize);

  if (E.syntax == NULL) return;

  char \*\*keywords = E.syntax->keywords;

  char \*scs = E.syntax->singleline\_comment\_start;

  char \*mcs = E.syntax->multiline\_comment\_start;

  char \*mce = E.syntax->multiline\_comment\_end;

  int scs\_len = scs ? strlen(scs) : 0;

  int mcs\_len = mcs ? strlen(mcs) : 0;

  int mce\_len = mce ? strlen(mce) : 0;

  int prev\_sep = 1;

  int in\_string = 0;

  int in\_comment = (row->idx > 0 && E.row[row->idx - 1].hl\_open\_comment);

  int i = 0;

  while (i < row->rsize) {

    char c = row->render[i];

    unsigned char prev\_hl = (i > 0) ? row->hl[i - 1] : HL\_NORMAL;

    if (scs\_len && !in\_string && !in\_comment) {

      if (!strncmp(&row->render[i], scs, scs\_len)) {

        memset(&row->hl[i], HL\_COMMENT, row->rsize - i);

        break;

      }

    }

    if (mcs\_len && mce\_len && !in\_string) {

      if (in\_comment) {

        row->hl[i] = HL\_MLCOMMENT;

        if (!strncmp(&row->render[i], mce, mce\_len)) {

          memset(&row->hl[i], HL\_MLCOMMENT, mce\_len);

          i += mce\_len;

          in\_comment = 0;

          prev\_sep = 1;

          continue;

        } else {

          i++;

          continue;

        }

      } else if (!strncmp(&row->render[i], mcs, mcs\_len)) {

        memset(&row->hl[i], HL\_MLCOMMENT, mcs\_len);

        i += mcs\_len;

        in\_comment = 1;

        continue;

      }

    }

    if (E.syntax->flags & HL\_HIGHLIGHT\_STRINGS) {

      if (in\_string) {

        row->hl[i] = HL\_STRING;

        if (c == '\\' && i + 1 < row->rsize) {

          row->hl[i + 1] = HL\_STRING;

          i += 2;

          continue;

        }

        if (c == in\_string) in\_string = 0;

        i++;

        prev\_sep = 1;

        continue;

      } else {

        if (c == '"' || c == '\'') {

          in\_string = c;

          row->hl[i] = HL\_STRING;

          i++;

          continue;

        }

      }

    }

    if (E.syntax->flags & HL\_HIGHLIGHT\_NUMBERS) {

      if ((isdigit(c) && (prev\_sep || prev\_hl == HL\_NUMBER)) ||

          (c == '.' && prev\_hl == HL\_NUMBER)) {

        row->hl[i] = HL\_NUMBER;

        i++;

        prev\_sep = 0;

        continue;

      }

    }

    if (prev\_sep) {

      int j;

      for (j = 0; keywords[j]; j++) {

        int klen = strlen(keywords[j]);

        int kw2 = keywords[j][klen - 1] == '|';

        if (kw2) klen--;

        if (!strncmp(&row->render[i], keywords[j], klen) &&

            is\_separator(row->render[i + klen])) {

          memset(&row->hl[i], kw2 ? HL\_KEYWORD2 : HL\_KEYWORD1, klen);

          i += klen;

          break;

        }

      }

      if (keywords[j] != NULL) {

        prev\_sep = 0;

        continue;

      }

    }

    prev\_sep = is\_separator(c);

    i++;

  }

  int changed = (row->hl\_open\_comment != in\_comment);

  row->hl\_open\_comment = in\_comment;

  if (changed && row->idx + 1 < E.numrows)

    editorUpdateSyntax(&E.row[row->idx + 1]);

}

//Done

int editorSyntaxToColor(int hl){

    switch(hl){

        case HL\_COMMENT:

        case HL\_MLCOMMENT: return 36;

        case HL\_KEYWORD1: return 33;

        case HL\_KEYWORD2: return 32;

        case HL\_STRING: return 35;

        case HL\_NUMBER: return 31;

        case HL\_MATCH: return 34;

        default: return 37;

    }

}

void editorSelectSyntaxHighlighting(){

    E.syntax = NULL;

    if(E.filename == NULL)return;

    char \*ext = strrchr(E.filename, '.');

    for(unsigned int j = 0; j < HLDB\_ENTRIES; j++){

        struct editorSyntax \*s = &HLDB[j];

        unsigned int i = 0;

        while(s->filematch[i]){

            int is\_ext = (s->filematch[i][0] == '.');

            if((is\_ext && ext && !strcmp(ext, s->filematch[i])) || (!is\_ext && strstr(E.filename, s->filematch[i]))){

                E.syntax = s;

                int filerow;

                for(filerow = 0; filerow < E.numrows; filerow++){

                    editorUpdateSyntax(&E.row[filerow]);

                }

                return;

            }

            i++;

        }

    }

}

/\*\* row operations \*\*/

int editorRowCxToRx(erow \*row, int cx){

    int rx = 0;

    int j;

    for(j = 0; j < cx; j++){

        if(row->chars[j] == '\t'){

            rx += (KILO\_TAB\_STOP - 1) - (rx % KILO\_TAB\_STOP);

        }

        rx++;

    }

    return rx;

}

int editorRowRxToCx(erow \*row, int rx){

    int cur\_rx = 0;

    int cx;

    for(cx = 0; cx < row->size; cx++){

        if(row->chars[cx] == '\t')

            cur\_rx += (KILO\_TAB\_STOP - 1) - (cur\_rx % KILO\_TAB\_STOP);

        cur\_rx++;

        if(cur\_rx > rx) return cx;

    }

    return cx;

}

void editorUpdateRow(erow \*row){

    int tabs = 0;

    int j;

    for(j = 0; j < row-> size;j++){

        if(row->chars[j] == '\t') tabs++;

    }

    free(row->render);

    row->render = malloc(row->size+ tabs\*(KILO\_TAB\_STOP - 1) + 1);

    int idx = 0;

    for(j = 0; j < row->size; j++){

        if(row-> chars[j] == '\t'){

            row->render[idx++] = ' ';

            while(idx % KILO\_TAB\_STOP != 0) row->render[idx++] = ' ';

        }else{

            row->render[idx++] = row->chars[j];

        }

    }

    row->render[idx] = '\0';

    row->rsize = idx;

    editorUpdateSyntax(row);

}

//Done

void editorInsertRow(int at, char \*s, size\_t len){

    if(at < 0 || at > E.numrows) return;

    E.row = realloc(E.row, sizeof(erow) \* (E.numrows+1));

    memmove(&E.row[at + 1], &E.row[at], sizeof(erow) \* (E.numrows - at));

    for(int j = at + 1; j <= E.numrows; j++) E.row[j].idx++;

    E.row[at].idx = at;

    E.row[at].size = len;

    E.row[at].chars = malloc(len + 1);

    memcpy(E.row[at].chars,s,len);

    E.row[at].chars[len] = '\0';

    E.row[at].rsize = 0;

    E.row[at].render = NULL;

    E.row[at].hl = NULL;

    E.row[at].hl\_open\_comment = 0;

    editorUpdateRow(&E.row[at]);

    E.numrows++;

    E.dirty++;

}

void editorFreeRow(erow \*row){

    free(row->render);

    free(row->chars);

    free(row->hl);

}

//Done

void editorDelRow(int at){

    if(at < 0 || at >= E.numrows)return;

    editorFreeRow(&E.row[at]);

    memmove(&E.row[at], &E.row[at+1], sizeof(erow) \* (E.numrows - at - 1));

    for(int j = at; j < E.numrows - 1; j++) E.row[j].idx--;

    E.numrows--;

    E.dirty ++;

}

void editorRowInsertChar(erow \*row, int at, int c){

    if(at < 0 || at > row->size) at = row->size;

    row->chars = realloc(row->chars, row->size + 2);

    memmove(&row->chars[at+1], &row->chars[at], row->size - at + 1);

    row->size++;

    row->chars[at] = c;

    editorUpdateRow(row);

    E.dirty++;

}

void editorRowAppendString(erow \*row, char \*s, size\_t len){

    row->chars = realloc(row->chars, row->size + len + 1);

    memcpy(&row->chars[row->size], s, len);

    row->size += len;

    row->chars[row->size] = '\0';

    editorUpdateRow(row);

    E.dirty++;

}

void editorRowDelChar(erow \*row, int at){

    if(at < 0 || at >= row->size)return;

    memmove(&row->chars[at], &row->chars[at + 1], row->size - at);

    row->size--;

    editorUpdateRow(row);

    E.dirty++;

}

/\*\* editor operations \*\*/

void editorInsertChar(int c){

    if(E.cy == E.numrows){

        editorInsertRow(E.numrows, "",0);

    }

    editorRowInsertChar(&E.row[E.cy], E.cx, c);

    E.cx++;

}

void editorInsertNewLine(){

    if(E.cx == 0){

        editorInsertRow(E.cy, "", 0);

    } else {

        erow \*row = &E.row[E.cy];

        editorInsertRow(E.cy + 1, &row->chars[E.cx], row->size - E.cx);

        row = &E.row[E.cy];

        row->size = E.cx;

        row->chars[row->size] = '\0';

        editorUpdateRow(row);

    }

    E.cy++;

    E.cx = 0;

}

void editorDelChar(){

    if (E.cy == E.numrows) return;

    if(E.cx == 0 && E.cy == 0) return;

    erow \*row = &E.row[E.cy];

    if(E.cx > 0){

        editorRowDelChar(row, E.cx -1);

        E.cx--;

    }else{

        E.cx = E.row[E.cy - 1].size;

        editorRowAppendString(&E.row[E.cy - 1], row->chars, row->size);

        editorDelRow(E.cy);

        E.cy--;

    }

}

/\*\* file i/o \*\*\*/

char \*editorRowToString(int \*buflen){

    int totlen = 0;

    int j;

    for(j = 0; j < E.numrows; j++){

        totlen += E.row[j].size+1;

    }

    \*buflen = totlen;

    char \* buf = malloc(totlen);

    char \*p = buf;

    for(j = 0; j < E.numrows; j++){

        memcpy(p, E.row[j].chars, E.row[j].size);

        p += E.row[j].size;

        \*p = '\n';

        p++;

    }

    return buf;

}

void editorOpen(char \* filename) {

    free(E.filename);

    E.filename = strdup(filename);

    editorSelectSyntaxHighlighting();

    FILE \*fp = fopen(filename, "r");

    if(!fp) die("fopen");

    char \*line = NULL;

    size\_t linecap = 0;

    ssize\_t linelen;

    while ((linelen = getline(&line, &linecap, fp)) != -1) {

        while(linelen > 0 && (line[linelen - 1] =='\n' || line[linelen - 1] == '\r'))

            linelen--;

       editorInsertRow(E.numrows, line, linelen);

    }

    free(line);

    fclose(fp);

    E.dirty = 0;

}

void editorSave(){

    if(E.filename == NULL) {

        E.filename = editorPrompt("Save as :%s (ESC to cancel)", NULL);

        if(E.filename == NULL){

            editorSetStatusMessage("Save Aborted");

            return;

        }

        editorSelectSyntaxHighlighting();

    }

    int len;

    char \*buf = editorRowToString(&len);

    int fd = open(E.filename, O\_RDWR | O\_CREAT, 0644);

    if(fd != -1){

        if(ftruncate(fd,len) != -1){

            if(write(fd, buf, len) == len){

                close(fd);

                free(buf);

                E.dirty = 0;

                editorSetStatusMessage("%d bytes written to disk", len);

                return;

            }

        }

        close(fd);

    }

    free(buf);

    editorSetStatusMessage("Cant save! I/O error: %s", strerror(errno));

}

/\*\* editor find \*\*/

void editorFindCallback(char \*query, int key){

    static int last\_match = -1;

    static int direction = 1;

    static int saved\_hl\_line;

    static char \*saved\_hl = NULL;

    if(saved\_hl){

        memcpy(E.row[saved\_hl\_line].hl, saved\_hl, E.row[saved\_hl\_line].rsize);

        free(saved\_hl);

        saved\_hl = NULL;

    }

    if(key == '\r' || key == '\x1b'){

        last\_match = -1;

        direction = 1;

        return;

    }else if(key == ARROW\_RIGHT || key == ARROW\_DOWN){

        direction = 1;

    }else if(key == ARROW\_LEFT || key == ARROW\_RIGHT) {

        direction = 01;

    }else {

        last\_match = -1;

        direction = 1;

    }

    if(last\_match == -1)direction = 1;

    int current = last\_match;

    int i;

    for(i = 0; i < E.numrows; i++){

        current += direction;

        if(current == -1) current = E.numrows-1;

        else if(current == E.numrows)current = 0;

        erow \*row = &E.row[current];

        char \*match = strstr(row->render, query);

        if(match){

            last\_match = current;

            E.cy = current;

            E.cx = editorRowRxToCx(row, match - row->render);

            E.rowoff = E.numrows;

            saved\_hl\_line = current;

            saved\_hl = malloc(row->rsize);

            memcpy(saved\_hl, row->hl, row->rsize);

            memset(&row->hl[match - row->render], HL\_MATCH, strlen(query));

            break;

        }

    }

}

void editorFind(){

    int saved\_cx = E.cx;

    int saved\_cy = E.cy;

    int saved\_coloff = E.coloff;

    int saved\_rowoff = E.rowoff;

    char \*query = editorPrompt("Search: %s (Use ESC/Arrows/Enter)", editorFindCallback);

    if(query){

        free(query);

    }else{

        E.cx = saved\_cx;

        E.cy = saved\_cy;

        E.coloff = saved\_coloff;

        E.rowoff = saved\_rowoff;

    }

}

/\*\* append buffer \*\*/

struct abuf{

    char \*b;

    int len;

};

#define ABUF\_INIT {NULL, 0}

void abAppend(struct abuf \*ab, const char \*s, int len){

    char \*new = realloc(ab->b, ab->len + len);

    if(new == NULL) return;

    memcpy(&new[ab->len], s, len);

    ab->b = new;

    ab->len += len;

}

void abFree(struct abuf \*ab){

    free(ab->b);

}

/\*\* input \*\*/

char \*editorPrompt(char \*prompt, void(\*callback)(char \*, int)){

    size\_t bufsize = 128;

    char \*buf = malloc(bufsize);

    size\_t buflen = 0;

    buf[0] = '\0';

    while(1){

        editorSetStatusMessage(prompt, buf);

        editorRefreshScreen();

        int c = editorReadKey();

        if(c == DEL\_KEY || c == CTRL\_KEY('h') || c == BACKSPACE){

            if(buflen != 0) buf[--buflen] = '\0';

        }else if(c == '\x1b'){

            editorSetStatusMessage("");

            if(callback) callback(buf,c);

            free(buf);

            return NULL;

        }else if(c == '\r'){

            if(buflen != 0){

                editorSetStatusMessage("");

                if(callback) callback(buf,c);

                return buf;

            }

        } else if( !iscntrl(c) && c < 128){

            if(buflen == bufsize -1){

                bufsize \*= 2;

                buf = realloc(buf, bufsize);

            }

            buf[buflen++] = c;

            buf[buflen] = '\0';

        }

        if(callback) callback(buf,c);

    }

}

void editorMoveCursor(int key){

    erow \*row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];

    switch(key){

        case ARROW\_LEFT:

            if(E.cx != 0){

                E.cx--;

            } else if(E.cy > 0){

                E.cy--;

                E.cx = E.row[E.cy].size;

            }

            break;

        case ARROW\_RIGHT:

            if(row && E.cx < row->size){

                 E.cx++;

            } else if (row && E.cx == row->size){

                E.cy++;

                E.cx = 0;

            }

            break;

        case ARROW\_UP:

            if(E.cy != 0){

                E.cy--;

            }

            break;

        case ARROW\_DOWN:

            if(E.cy < E.numrows){

                E.cy++;

            }

            break;

    }

    row = (E.cy >= E.numrows) ? NULL : &E.row[E.cy];

    int rowlen = row ? row->size : 0;

    if(E.cx > rowlen){

        E.cx = rowlen;

    }

}

void editorProcessKeypress(){

    static int quit\_times = KILO\_QUIT\_TIMES;

    int c = editorReadKey();

    switch(c){

        case '\r':

            editorInsertNewLine();

            break;

        case CTRL\_KEY('q'):

            if(E.dirty && quit\_times > 0){

                editorSetStatusMessage("WARNING!! File has unsaved changes. Press CTRL-Q %d more times to quit",

                    quit\_times--);

                return;

            }

            write(STDOUT\_FILENO, "\x1b[2J", 4);

            write(STDOUT\_FILENO, "\x1b[H",3);

            exit(0);

            break;

        case CTRL\_KEY('s'):

            editorSave();

            break;

        case HOME\_KEY:

            E.cx = 0;

            break;

        case END\_KEY:

            if(E.cy < E.numrows)

                E.cx = E.row[E.cy].size;

            break;

        case CTRL\_KEY('f'):

            editorFind();

            break;

        case BACKSPACE:

        case CTRL\_KEY('h'):

        case DEL\_KEY:

            if(c == DEL\_KEY)editorMoveCursor(ARROW\_RIGHT);

            editorDelChar();

            break;

        case PAGE\_UP:

        case PAGE\_DOWN:

            {

                if(c == PAGE\_UP){

                    E.cy = E.rowoff;

                }else if(c == PAGE\_DOWN){

                    E.cy = E.rowoff + E.screenrows - 1;

                    if(E.cy > E.numrows) E.cy = E.numrows;

                }

                int times = E.screenrows;

                while(times--)

                    editorMoveCursor(c == PAGE\_UP ? ARROW\_UP : ARROW\_DOWN);

            }

            break;

        case ARROW\_UP:

        case ARROW\_LEFT:

        case ARROW\_DOWN:

        case ARROW\_RIGHT:

            editorMoveCursor(c);

            break;

        case CTRL\_KEY('l'):

        case '\x1b':

            break;

        default:

            editorInsertChar(c);

            break;

    }

    quit\_times = KILO\_QUIT\_TIMES;

}

/\*\* output \*\*/

void editorScroll() {

  E.rx = 0;

  if (E.cy < E.numrows) {

    E.rx = editorRowCxToRx(&E.row[E.cy], E.cx);

  }

  if (E.cy < E.rowoff) {

    E.rowoff = E.cy;

  }

  if (E.cy >= E.rowoff + E.screenrows) {

    E.rowoff = E.cy - E.screenrows + 1;

  }

  if (E.rx < E.coloff) {

    E.coloff = E.rx;

  }

  if (E.rx >= E.coloff + E.screencols) {

    E.coloff = E.rx - E.screencols + 1;

  }

}

//Done

void editorDrawRows(struct abuf \*ab) {

  int y;

  for (y = 0; y < E.screenrows; y++) {

    int filerow = y + E.rowoff;

    if (filerow >= E.numrows) {

      if (E.numrows == 0 && y == E.screenrows / 3) {

        char welcome[80];

        int welcomelen = snprintf(welcome, sizeof(welcome),

          "Kilo editor -- version %s", KILO\_VERSION);

        if (welcomelen > E.screencols) welcomelen = E.screencols;

        int padding = (E.screencols - welcomelen) / 2;

        if (padding) {

          abAppend(ab, "~", 1);

          padding--;

        }

        while (padding--) abAppend(ab, " ", 1);

        abAppend(ab, welcome, welcomelen);

      } else {

        abAppend(ab, "~", 1);

      }

    } else {

      int len = E.row[filerow].rsize - E.coloff;

      if (len < 0) len = 0;

      if (len > E.screencols) len = E.screencols;

      char \*c = &E.row[filerow].render[E.coloff];

      unsigned char \*hl = &E.row[filerow].hl[E.coloff];

      int current\_color = -1;

      int j;

      for (j = 0; j < len; j++) {

        if (iscntrl(c[j])) {

          char sym = (c[j] <= 26) ? '@' + c[j] : '?';

          abAppend(ab, "\x1b[7m", 4);

          abAppend(ab, &sym, 1);

          abAppend(ab, "\x1b[m", 3);

          if (current\_color != -1) {

            char buf[16];

            int clen = snprintf(buf, sizeof(buf), "\x1b[%dm", current\_color);

            abAppend(ab, buf, clen);

          }

        } else if (hl[j] == HL\_NORMAL) {

          if (current\_color != -1) {

            abAppend(ab, "\x1b[39m", 5);

            current\_color = -1;

          }

          abAppend(ab, &c[j], 1);

        } else {

          int color = editorSyntaxToColor(hl[j]);

          if (color != current\_color) {

            current\_color = color;

            char buf[16];

            int clen = snprintf(buf, sizeof(buf), "\x1b[%dm", color);

            abAppend(ab, buf, clen);

          }

          abAppend(ab, &c[j], 1);

        }

      }

      abAppend(ab, "\x1b[39m", 5);

    }

    abAppend(ab, "\x1b[K", 3);

    abAppend(ab, "\r\n", 2);

  }

}

void editorDrawStatusBar(struct abuf \*ab){

    abAppend(ab, "\x1b[7m", 4);

    char status[80], rstatus[80];

    int len = snprintf(status, sizeof(status), "%.20s - %d lines %s",

        E.filename ? E.filename : "[No Name]", E.numrows,

        E.dirty ? "(modified)" : "");

    int rlen = snprintf(rstatus, sizeof(rstatus), "%s | %d%d", E.syntax ? E.syntax->filetype: "no ft", E.cy + 1, E.numrows);

    if(len > E.screencols)len = E.screencols;

    abAppend(ab, status,len);

    while(len < E.screencols){

        if(E.screencols - len == rlen){

            abAppend(ab, rstatus, rlen);

            break;

        }else{

            abAppend(ab, " ", 1);

            len++;

        }

    }

    abAppend(ab, "\x1b[m", 3);

    abAppend(ab, "\r\n", 2);

}

void editorDrawMessageBar(struct abuf \*ab){

    abAppend(ab, "\x1b[K" , 3);

    int msglen = strlen(E.statusmsg);

    if(msglen > E.screencols) msglen = E.screencols;

    if(msglen && time(NULL) - E.statusmsg\_time < 5)

        abAppend(ab, E.statusmsg, msglen);

}

void editorRefreshScreen(){

    editorScroll();

    struct abuf ab = ABUF\_INIT;

    abAppend(&ab, "\x1b[?25l", 6);

    abAppend(&ab, "\x1b[H", 3);

    editorDrawRows(&ab);

    editorDrawStatusBar(&ab);

    editorDrawMessageBar(&ab);

    char buf[32];

    snprintf(buf, sizeof(buf), "\x1b[%d;%dH", (E.cy - E.rowoff) + 1, (E.rx - E.coloff) + 1);

    abAppend(&ab, buf, strlen(buf));

    abAppend(&ab, "\x1b[?25h", 6);

    write(STDOUT\_FILENO, ab.b, ab.len);

    abFree(&ab);

}

void editorSetStatusMessage(const char \*fmt, ...){

    va\_list ap;

    va\_start(ap,fmt);

    vsnprintf(E.statusmsg, sizeof(E.statusmsg), fmt, ap);

    va\_end(ap);

    E.statusmsg\_time = time(NULL);

}

/\*\* init \*\*/

void initEditor(){

    E.cx = 0;

    E.cy = 0;

    E.rx = 0;

    E.rowoff = 0;

    E.coloff = 0;

    E.numrows = 0;

    E.row = NULL;

    E.dirty = 0;

    E.filename = NULL;

    E.statusmsg[0] = '\0';

    E.statusmsg\_time = 0;

    E.syntax = NULL;

    if(getWindowSize(&E.screenrows, &E.screencols) == -1) die("getWindowSize");

    E.screenrows -= 2;

}

int main(int argc, char\* argv[]){

    enableRawMode();

    initEditor();

    if(argc >= 2){

        editorOpen(argv[1]);

    }

    editorSetStatusMessage("HELP: CTRL-S = save | CTRL-Q = quit | CTRL-F = find");

    while(1){

        editorRefreshScreen();

        editorProcessKeypress();

    }

    return 0;

}