```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file alias.c
 * @brief Definition of alias functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-08-02
 */
#include "alias.h"
#include "util.h"

/**
 * @brief Adds an alias to the alias list.
 *
 * Inserts an alias into the linked list structure that stores aliases
 * (::aliasList).  A call to detokenize() is made since aliases are stored as
 * unparsed commands.  Any existing alias with the same name will be overwritten
 * when this function is called.
 *
 * @param env           A pointer to the global ::kgenv object.
 * @param name          The name of the alias.
 * @param cmd_argc      The argument count for the command the alias points to.
 * @param cmd_argv[]    The argument values for the command the alias points to.
 */
void add_alias(kgenv* env, char* name, int cmd_argc, char* cmd_argv[]){

    // Allocate space for the new alias
    aliasList* new_alias = malloc(sizeof(aliasList));
    if(new_alias == NULL){
        perror("Failed to add alias");
        return;
    }

    // Delete any existing alias with the same name
    remove_alias(env, name);

    // Copy over the alias name
    new_alias->name = (char*)malloc(strlen(name) + 1);
    if(new_alias->name == NULL){
        perror("Failed to add alias");
        return;
    }
    strcpy(new_alias->name, name);

    // Copy over the argv arrray and reconstruct the command line string so
    // the recursive calls work out correctly when processing the alias.  We're
    // doing some unecessary processing using this method, but the
    // implementation is neater.
    int line_length = 0;
    for(int i=0; i < cmd_argc; i++){
        line_length += strlen(cmd_argv[i]);
        line_length++;  // For null character
    }

    new_alias->string = malloc(line_length);
    if(new_alias->string == NULL){
        perror("Failed to add alias");
        return;
    }

    memcpy(new_alias->string , *cmd_argv, line_length);
```

```c
    detokenize(new_alias->string, line_length);

    // Add the link to the next node
    new_alias->next = env->aliases;


    env->aliases = new_alias;
}



/**
 * @brief Checks if a command entered is an alias.
 *
 * @param env   A pointer to the global ::kgenv object.
 * @param name  The name to check.  This should be argv[0] of the command that's
 * been entered.  This function does not parse an entire command line.
 *
 * @return  If an alias exists with the name, a pointer to the ::aliasList node
 * where the alias is stored is returned.  Otherwise NULL is returned if the
 * alias does not exist.
 */
aliasList* is_alias(kgenv* env, char* name){
    aliasList* a = env->aliases;

    while(a != NULL){

        if(strcmp(name, a->name) == 0){
            return a;
        }

        a = a->next;
    }
    return NULL;
}

/**
 * @brief Removes an alias if it exists.
 *
 * Steps through the alias list (::aliasList) stored in the global ::kgenv
 * object.  If an alias of the specified name is found, it is removed from the
 * list.  No action is taken if an alias with the name is not found.
 *
 * @param env   The global ::kgenv environment object.
 * @param name  The name of the alias to remove.
 *
 * @return  True if an alias was removed.  False otherwise.
 */
bool remove_alias(kgenv* env, char* name){

    aliasList* a = env->aliases;
    aliasList* prev = NULL;

    while(a != NULL){

        if(strcmp(name, a->name) == 0){

            if(prev != NULL){
                prev->next = a->next;
            } else {
                env->aliases = a->next;
            }

            free(a->name);
            free(a->string);
            free(a);
            return true;

        }
        prev = a;
        a = a->next;
    }

    return false;
}
```

```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file builtins.c
 * @brief Definitions of builtin functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009−08−02
 */
#include "builtins.h"
#include "get_path.h"
#include "alias.h"
#include "wildcard.h"
#include "watchmail.h"

#include <readline/readline.h>

extern int errno;

//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//−− The following constants define the built−in commands.  The commands are
//−− matched to the function pointers with a one−to−one matching done in order.
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

/**
 * @brief Stores the commands that map to the built−in functions.
 *
 * These strings are what, if entered as the zeroth argument (argv[0]) in a
 * command will execute a built−in function.
 */
const char* BUILT_IN_COMMANDS[] = {
    "exit",
    "which",
    "where",
    "cd",
    "pwd",
    "list",
    "pid",
    "kill",
    "prompt",
    "printenv",
    "alias",
    "unalias",         // Not a requirement, but easy to add.
    "history",
    "setenv",
    "lsbuiltins",
    "watchmail",
    "noclobber",
    "vimode",
    "emacsmode"
#ifdef DEBUG        // Various built ins defined for debugging purposes.
    ,
    "_db_tokenizer",
    "_db_kgenv",
    "_db_path",
    "_db_history",
    "_db_wc_contains",
    "_db_wc_expand"
#endif //DEBUG
};


/**
```

```c
 * @brief An array of function pointers for built−in commands.
 
 * These function pointers map one−to−one in order with the command strings in
 * ::BUILT_IN_COMMANDS.  Each built−in command function has the same prototype.
 * Setting the prototypes up in this way allows us to write each built−in as if
 * it were a "main" functions of a seperate program with access to the ::kgenv
 * structure.  Adding new built−ins is very easy.
 *
 * @param env The global ::kgenv structure is the first argument to every
 * built−in command.
 * @param argc The second argument is always the argument count of the command
 * being proccessed.
 * @param argv The third argument is always the argument value array of the
 * command being processed.
 */
void (*BUILT_IN_FUNCS[])(kgenv* env, int argc, char** argv) = {
    bic_exit,
    bic_which,
    bic_where,
    bic_cd,
    bic_pwd,
    bic_list,
    bic_pid,
    bic_kill,
    bic_prompt,
    bic_printenv,
    bic_alias,
    bic_unalias,
    bic_history,
    bic_setenv,
    bic_lsbuiltins,
    bic_watchmail,
    bic_noclobber,
    bic_vimode,
    bic_emacsmode
#ifdef DEBUG        // various built ins defined for debugging purposes
        ,
    _db_tokenizer,
    _db_kgenv,
    _db_path,
    _db_history,
    _db_wc_contains,
    _db_wc_expand
#endif //DEBUG
};


/**
 * @brief Checks if a command is a built−in command.
 *
 * Loops through ::BUILT_IN_COMMANDS comparing the command parameter to each
 * string of ::BUILT_IN_COMMANDS.  When a match is found the index plus one is
 * returned.
 *
 * @param command The command to check.
 *
 * @return Returns 0 if the command is not built−in, and a positive value that
 * is one greather than the index of the function in the built−in definitions
 * arrays (::BUILT_IN_COMMANDS and ::BUILT_IN_FUNCS) if the command is built−in.
 */
short int is_builtin(char* command){
    for(int i=0; i < NUM_BUILTINS; i++){
    int result = strcmp(command, BUILT_IN_COMMANDS[i]);
    if(result == 0)
        return i + 1;
    }

    return 0;
}
```

```c
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//−− Definitions of the various built in functions.
//−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

/**
 * @brief Built−in exit command.
 *
 * Exits with status 0;
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_exit(kgenv* env, int argc, char* argv[]){
    exit(0);
}

/**
 * @brief Built−in which command.
 *
 * Displays the full path to the executable that will be executed for each
 * command that is given as an argument.  The path printed is the first one that
 * occurs in the PATH environment variable that contains a file of the correct
 * name with execute permissions.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_which(kgenv* env, int argc, char* argv[]){
    if(argc == 1){
    fprintf(stderr, "which: too few arguments\n");
    return;
    }

    // Loop through each argument and display the path
    for(int i = 1; i < argc; i++){
    char* path = which(argv[i], env−>path);

    if(path != NULL){
        printf("%s\n", path);
        free(path);
    }
    }
}

/**
 * @brief Built−in where command.
 *
 * Same as the which command, but displays all of the possible paths where a
 * file of the correct name with executable permissions exist within the PATH
 * envrionment variable list of paths.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_where(kgenv* env, int argc, char* argv[]){
    if(argc == 1){
    fprintf(stderr, "where: Too few arguments.\n");
    return;
    }

    // For loop executed once for each argument
    for(int i = 1; i < argc; i++){
    pathList* pl = env−>path;
    char* cmd = argv[i];
```

```c
    // While loop executed once for ach directory in the path
    while(pl != NULL){
        DIR* dirp = opendir(pl−>element);
        if(dirp == NULL){
        perror("Error in where");
        return;
        } else {
        struct dirent* dp = readdir(dirp); //TODO: check errno?

        // While loop executed once for each file in directory
        while(dp != NULL){
            if(strcmp(dp−>d_name, cmd) == 0){
            printf("%s/%s\n", pl−>element, cmd);
            }
            dp = readdir(dirp);
        }

        if(closedir(dirp) == −1){
            perror("Error in where");
            return;
        }
        }
        pl = pl−>next;
    }
    }
}

/**
 * @brief Built−in cd command.
 *
 * Changes the current working directory using the chdir library function.  When
 * called with no arguments, changes to the user’s home directory.  When called
 * as "cd −", changes to the previous directory.
 *
 * Before switching, the previous directory and the current directory are set in
 * the global ::kgenv environment object.  The user’s home directory is also
 * retrievable from this object.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_cd(kgenv* env, int argc, char* argv[]){
    //** Does nothing if executed with more than one argument
    if(argc > 2){
    fprintf(stderr, "cd: Too many arguments.\n");
    return;
    }

    //** When called with no arguments cd to home directory
    if(argc == 1){

    // Free up the previous directory and set previous to current
    if(env−>pwd != NULL) free(env−>pwd);
    env−>pwd = env−>cwd;

    // Set the current directory to the home directory
    env−>cwd = (char*)malloc(strlen(env−>homedir) + 1);
    if(env−>cwd == NULL){
        perror("Error in cd");
        return;
    }
    strcpy(env−>cwd, env−>homedir);

    // Use chdir to change the working directory
    if(chdir(env−>cwd) != 0) {
        perror("Error in chdir");
    }
    }

    else

    //** If called as "cd -", cd to the previous directory (pwd in kgenv)
    if(strcmp(argv[1], "-") == 0){

    // Swap the current working directory with the previous working
    // directory
    char* temp = env->cwd;
    env->cwd = env->pwd;
    env->pwd = temp;

    // Use chdir to change the working directory
    if(chdir(env->cwd) != 0) {
        perror("Error in chdir");
    }

    }

    else

    //** Otherwise we have either a relative or absolute path to a directory
    {
    // Change to the path specified in the argument
    if(chdir(argv[1]) != 0){
        perror("Error in chdir");
        return;
    }

    // Free up the previous directory and set previous to current
    if(env->pwd != NULL) free(env->pwd);
    env->pwd = env->cwd;

    // Set the current working directory string.  Using getcwd allows us to
    // avoid having to resolve an absolute path if the argument is relative.
    env->cwd = getcwd(NULL, CWD_BUFFER_SIZE);
    }
}

/**
 * @brief Built-in pwd command.
 *
 * Prints the current working directory to stdout.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_pwd(kgenv* env, int argc, char* argv[]){

    // Print the current working directory
    printf("%s\n", env->cwd);

}

/**
 * @brief Built-in list command.
 *
 * Lists files in the directores specified as arguments.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
// TODO: reverse order of printout
void bic_list(kgenv* env, int argc, char* argv[]){
    DIR* dirp;            // directory pointer
    bool cwd_mode = false;    // true if passed with no args
```

```

    // If called with no arguments we just add an argument that is the current
    // working directory.
    if(argc == 1){
    argc++;
    argv[1] = env->cwd;
    cwd_mode = true;
    }


    // Loop over the argument list and print each directory listing.
    for(int i=1; i < argc; i++){
    // Only print the directory name if we are processing arguments.
    if(!cwd_mode)
        printf("\n%s:\n", argv[i]);

    dirp = opendir(argv[i]);
    if(dirp == NULL){
        perror("Error in list");
        return;

    } else {

        // This loop iterates through the directory stream.
        struct dirent* dp = readdir(dirp);    //TODO: check errno?
        while(dp != NULL){
        printf("%s\n", dp->d_name);
        dp = readdir(dirp);
        }

        if(closedir(dirp) == -1){
        perror("Error in list");
        return;
        }
    }

    }
}


/**
 * @brief Built-in pid command.
 *
 * Prints the process id (pid) of the shell.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_pid(kgenv* env, int argc, char* argv[]){
    pid_t pid = getpid();
    if(pid == -1){        //TODO: check error condition
    perror("Error in pid");
    return;
    }

    printf("%d\n", pid);
}


/**
 * @brief Built-in kill command.
 *
 * Sends a SIGTERM signal to the pid specified in the arguments.  If a -n is
 * passed, the signal number n is passed to the specified process.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_kill(kgenv* env, int argc, char* argv[]){
    int pid;            ///< PID of the process to send signal to
    int signal = SIGTERM;    ///< Default signal is SIGTERM

    errno = 0;

    // Called with no arguments
    if(argc == 1){
    fprintf(stderr, "kill: Too few arguments.\n");
    return;
    }

    if(argc == 2){        // Called with just a pid

        pid = atoi(argv[1]);

    if(errno != 0){
        perror("Error in kill");
        return;
    }

    } else if(argc == 3){    // Called with a signal specified

        pid = atoi(argv[2]);
        signal = atoi(argv[1] + 1);    // Add one to remove hyphen

        if(errno != 0){
            perror("Error in kill");
            return;
        }

    } else {            // Called with too many arguments
        fprintf(stderr, "kill: Too many arguments.\n");
        return;
    }

    sigsend(P_PID, pid, signal);
    //printf("Sending code %d to pid %d\n", signal, pid);

    // Send the kill signal
    if(kill(pid, signal) == −1){
        perror("Error in kill");
    }

}


/**
 * @brief Built−in prompt command.
 *
 * Changes the prompt prefix to the specified argument.  If no argument is
 * passed, prompts the user for a prefix.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_prompt(kgenv* env, int argc, char* argv[]){
    char* new_prompt;        // the new prompt string

    // Case where we are passed arguments.
    if(argc > 1){
    new_prompt = argv[1];
    strcpy(env−>prompt, new_prompt);
    return;
    }

    // Case where we prompt user for input.
```

```
    printf("New prompt prefix: ");
    char* prompt_in = (char*)malloc(LINE_BUFFER_SIZE);
    if(prompt_in == NULL){
    perror("Error in prompt");
    return;
    }

    fgets(prompt_in, LINE_BUFFER_SIZE, stdin);

    // Need to remove trailing newline from input.
    if(prompt_in[strlen(prompt_in) − 1] == ’\n’){
    prompt_in[strlen(prompt_in) − 1] = ’\0’;
    }

    // Save some heap by re−allocating only what’s needed.
    new_prompt = (char*)malloc(strlen(prompt_in) + 1);
    if(new_prompt == NULL){
    perror("Error in prompt");
    return;
    }

    strcpy(new_prompt, prompt_in);
    env−>prompt = new_prompt;
    free(prompt_in);
}


/**
 * @brief Built−in printenv command.
 *
 * Prints out a list of environment variables and their values.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_printenv(kgenv* env, int argc, char* argv[]){

    // Called with no arguments, print entire environment
    if(argc == 1){
    char** i = environ;
    while(*i != NULL){
        printf("%s\n", *i);
        i++;
    }
    }

    // Called with one argument, print the value
    else if(argc == 2){
    char* value = getenv(argv[1]);
    if(value != NULL){
        printf("%s\n", value);
    } else {
        fprintf(stderr, "%s was not found in the current environment\n",
            argv[1]);
    }
    }


    // Called with more than one argument
    else {
    fprintf(stderr, "printenv: Too many arguments.\n");
    }

}


/**
 * @brief Built−in alias command.
 *
 * When run with no arguments prints a list of aliases currently in the alias
 * list.  When ran with arguments sets the alias name in the first argument to
 * the command specified in subsequent arguments.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_alias(kgenv* env, int argc, char* argv[]){

    // If no arguments are passed print the alias list
    if(argc == 1){
    aliasList* a = env->aliases;
    while(a != NULL){
        //TODO: update to print entire argv array
        printf("%s\t(%s)\n", a->name, a->string);
        a = a->next;
    }
    return;
    }

    // Add the alias to the list.  We need to decrement argc by 2 (command and
    // alias name).
    add_alias(env, argv[1], argc - 2, &argv[2]);
}


/**
 * @brief Built-in unalias command.
 *
 * Removes an alias from the alias list.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_unalias(kgenv* env, int argc, char* argv[]){
    //TODO: support multiple arguments
    if(argc == 2){
    remove_alias(env, argv[1]);
    }
}


/**
 * @brief Built-in history command.
 *
 * When run with no arguments, prints out the last 10 commands run.  When an
 * argument is passed, that number of commands is printed.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
//TODO: fix history
void bic_history(kgenv* env, int argc, char* argv[]){
    int num_items = 0;    // Number of commands to print

    // We default to printing 10 commands if no argument is passed
    if(argc == 1){
        num_items = 10;
    } else {
        errno = 0;
        num_items = atoi(argv[1]);
        if(errno != 0){
            perror("Error in history");
            return;
        }
```

```
    }

    // Output ordered pointers; we allocate space for num_items pointers even
    // if they aren't all going to be used.  Point to the histelement struct for
    // the given command.
    histList* outbuf[num_items];

    histList* h = env->hist;
    int j=num_items - 1;

    // Loop through the last
    while(h != NULL && j >= 0){
    outbuf[j] = h;
    h = h->next;
        j--;
    }

    j++;    // Need to increment j to adjust for final decrement
    for(int i=j; i < num_items; i++){
    printf("%d: %s\n", outbuf[i]->num, outbuf[i]->command);
    }
}


/**
 * @brief Built-in setenv command.
 *
 * When run with no arguments prints a list of environment variables and values.
 * When run with two arguments, sets the variable in the first argument equal to
 * the value in the second argument.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_setenv(kgenv* env, int argc, char* argv[]){

    // Called with no arguments, print entire environment
    if(argc == 1){
    char** i = environ;
        while(*i != NULL){
            printf("%s\n", *i);
            i++;
        }
    }

    // Called with one argument, set variable to null
    else if(argc == 2){
        set_environment(env, argv[1], "");
    }

    // Called with two arguments, set variable to 2nd argument
    else if(argc == 3){
        set_environment(env, argv[1], argv[2]);
    }

    // Called with too many arguments
    else {
        fprintf(stderr, "setenv: Too many arguments.\n");
    }

}

/**
 * @brief Built-in lsbuiltins command.
 *
 * Lists all built-in functions.  Ignores any arguments passed.
 * (Not a project requirement.)
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_lsbuiltins(kgenv* env, int argc, char* argv[]){

    for(int i=0; i < NUM_BUILTINS; i++){
        printf("%s\n", BUILT_IN_COMMANDS[i]);
    }

}

/**
 * @brief Built-in watchmail command.
 *
 * Watches for new mail in the specified file.  Prints a message and beeps when
 * mail is received.
 * //TODO: add more detailed documentation
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_watchmail(kgenv* env, int argc, char* argv[]){

    char* file;            //<< The file to watch

    int action;            //<< The new stop/start action on the file
    enum { START, STOP };    //<< Two states for the action variable


    //## Print out the help message if run with no arguments
    if(argc == 1 || argc > 3){
        printf("watchmail:\n\n\twatchmail [file] [on/off]\n\n");
        return;
    }

    file = (char*)malloc(strlen(argv[1]) + 1);    // Aliased in watchmails list
                        // (watch free)
    strcpy(file, argv[1]);

    //## If run with only one argument, we're starting a watchmail on the file
    if(argc == 2){
        action = START;
    }

    //## If run with two arguments, parse the second to find action needed
    else if(argc == 3){

    if(strcmp(argv[2], "off") == 0){
        action = STOP;
    } else if(strcmp(argv[2], "on") == 0){
        action = START;
    } else {
        fprintf(stderr, "watchmail: Invalid command.\n");
        return;
    }

    }


    //## Take the action to start/stop watchmail
    if(action == START){

        // Start watching the file
    printf("Starting watchmail for %s\n", file);
        control_watchmail(file, false, env);

    } else if(action == STOP){
```

```
        // Stop watching the file
    printf("Stopping watchmail for %s\n", file);
        control_watchmail(file, true, env);

    }

}

/**
 * @brief Built-in noclobber command.
 *
 * Changes the value of the clobber variable.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_noclobber(kgenv* env, int argc, char* argv[]){
    env->noclobber = !env->noclobber;
    printf("Clobbering is now %s. (value is %d)\n",
            env->noclobber ? "off" : "on",
            env->noclobber);
}

/**
 * @brief Sets command line editing to vi mode.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_vimode(kgenv* env, int argc, char* argv[]){
    rl_editing_mode = 0;
}

/**
 * @brief Sets command line editing to emacs mode.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void bic_emacsmode(kgenv* env, int argc, char* argv[]){
    rl_editing_mode = 1;
}

//-------------------------------------------------------------------------------
//-- Definitions of debug functions
//-------------------------------------------------------------------------------

#ifdef DEBUG

/**
 * @brief Debugs the tokenizer by showing argument count and argument values for
 * the arguments passed to ::_db_tokenizer.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void _db_tokenizer(kgenv* env, int argc, char* argv[]){
    printf("argc = %d\n", argc);
    for(int i=0; i<argc; i++){
        printf("argv[%d] = %s\n", i, argv[i]);
    }
}
```

```
/**
 * @brief Prints out some of the variables in the global ::kgenv environment
 * object for debugging purposes.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void _db_kgenv(kgenv* env, int argc, char* argv[]){

    printf("uid=%d\n", env->uid);
    printf("homedir=%s\n", env->homedir);

    printf("cwd=%s\n", env->cwd);
    printf("pwd=%s\n", env->pwd);

    printf("prompt=%s\n", env->prompt);
}


/**
 * @brief Prints out path list for debugging purposes.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void _db_path(kgenv* env, int argc, char* argv[]){
    pathList* p = env->path;
    while (p != NULL){
        printf("%s\n", p->element);
        p = p->next;
    }
}


/**
 * @brief Prints out entire history list for debugging purposes.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void _db_history(kgenv* env, int argc, char* argv[]){
    histList* h = env->hist;
    while (h != NULL){
        printf("%d: %s\n", h->num, h->command);
        h = h->next;
    }
}


/**
 * @brief Prints "true" if the first argument contains a wildcard and "false" if
 * it does not.  Used to debug ::contains_wildcards.
 *
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void _db_wc_contains(kgenv* env, int argc, char* argv[]){
    printf("%s\n", contains_wildcards(argv[1]) ? "true":"false");
}


/**
 * @brief Prints the expanded version of the first argument.  Used to debug
 * ::expand_argument.
 *
```

```
 * @param env A pointer to the global ::kgenv environment object.
 * @param argc The argument count for the command entered.
 * @param argv[] The argument values for the command entered.
 */
void _db_wc_expand(kgenv* env, int argc, char* argv[]){
    printf("%s\n", expand_argument(argv[1]));
}

#endif //DEBUG
```

```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/*
  get_path.c
  Ben Miller

  Just a little sample function that gets the PATH env var, parses it and
  puts it into a linked list, which is returned.
*/
#include "get_path.h"

struct pathelement *get_path()
{
  /* path is a copy of the PATH and p is a temp pointer */
  char *path, *p;

  /* tmp is a temp point used to create a linked list and pathlist is a
     pointer to the head of the list */
  struct pathelement *tmp, *pathlist = NULL;

  p = getenv("PATH");   /* get a pointer to the PATH env var.
                           make a copy of it, since strtok modifies the
                           string that it is working with... */
  path = malloc((strlen(p)+1)*sizeof(char));    /* use malloc(3C) this time */
  strncpy(path, p, strlen(p));
  path[strlen(p)] = '\0';

  p = strtok(path, ":");        /* PATH is : delimited */
  do                            /* loop through the PATH */
  {                             /* to build a linked list of dirs */
    if ( !pathlist )            /* create head of list */
    {
      tmp = calloc(1, sizeof(struct pathelement));
      pathlist = tmp;
    }
    else                        /* add on next element */
    {
      tmp->next = calloc(1, sizeof(struct pathelement));
      tmp = tmp->next;
    }
    tmp->element = p;
    tmp->next = NULL;
  } while ( p = strtok(NULL, ":") );

  return pathlist;
} /* end get_path() */
```

```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file ipc.c
 * @brief Implementations of IPC functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009−10−25
 */

#include "ipc.h"
#include "types.h"
#include "util.h"

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

const char* IPC_OPERATORS[] = { "|&", "|" };
const int NUM_IPC_OPERATORS = 5;

bool contains_ipc(char* line){
    return strstr(line, "|") || strstr(line, "|&");
}

enum ipc_opcodes parse_ipc_line(char** left, char** right, char* line){

    char* p = NULL;
    enum ipc_opcodes ipc_code;
    for(int i=0; i < NUM_IPC_OPERATORS && p == NULL; i++){
        p = strstr(line, IPC_OPERATORS[i]);
        ipc_code = i;
    }

    int left_length = (int)p − (int)line;
    *left = (char*)malloc(left_length + 1);
    memcpy(*left, line, left_length + 1);
    (*left)[left_length − 1] = ’\0’;

    char* ptr = strtok(line + left_length, "|&");
    int right_length = strlen(line) − (int)p + (int)line;
    *right = (char*)malloc(right_length + 1);
    memcpy(*right, ptr, right_length + 1);

    return ipc_code;
}

void perform_ipc(char* left, char* right, enum ipc_opcodes ipc_type,
        kgenv* env){

    int fid;
    int filedes[2];     //<< Index 0 will be a dup of stdin and index 1 a dup of
                        //<< stdout/stderr

    if(pipe(filedes) == −1){
        perror("Error creating pipe");
        return;
    }

    // Redirect stdin
    close(0);
    dup(filedes[0]);
    close(filedes[0]);
```

```c

    // Redirect stdout
    close(1);
    dup(filedes[1]);

    // Redirect stderr
    if(ipc_type == IPC_ALL){
        close(2);
        dup(filedes[1]);
    }

    close(filedes[1]);

    // Run the command on the left
    process_command_in(left, env, false, false);

    // Return stdout to terminal
    fid = open("/dev/tty", O_WRONLY);
    close(1);
    dup(fid);
    close(fid);

    // Return sterr to terminal
    fid = open("/dev/tty", O_WRONLY);
    close(2);
    dup(fid);
    close(fid);

    // Run the command on the right
    process_command_in(right, env, false, true);

    // Return stdin to terminal
    fid = open("/dev/tty", O_RDONLY);
    close(0);
    dup(fid);
    close(fid);
}
```

```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file kgsh.c
 * @brief kgsh main file
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-08-02
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <strings.h>
#include <limits.h>

#include <unistd.h>     // for access
#include <signal.h>

#include <pwd.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

#include <readline/readline.h>
#include <readline/history.h>

#include "builtins.h"
#include "util.h"
#include "get_path.h"

void initialize_environment(kgenv* env);

void sig_interrupt(int signal);

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

    kgenv global_env; // The global environment structure

    // Populate the global environment for the first time
    initialize_environment(&global_env);

    sigset(SIGINT, sig_interrupt);      // Interrupt function for Ctrl-C
    sigignore(SIGTSTP);                 // Ignore Ctrl-Z
    sigignore(SIGTERM);
    signal(SIGCHLD, SIG_IGN);

    char*  line_in = NULL; // Stores the command entered (pointed to by in_argv)

    // The main loop that is executed once for each command prompt.
    while(1){

        //## Print the shell prompt
        char* prompt = (char*)calloc(LINE_BUFFER_SIZE, sizeof(char));
        sprintf(prompt, "%s %s> ", global_env.prompt, global_env.cwd);


        //## Read the a line from the shell
        char* line_in = readline(prompt);

        //## Parse the command and execute the appropriate action
        process_command_in(line_in, &global_env, false, true);
        free(prompt);
```

```c
    }
}

/**
 * @brief Initializes the kgenv global environment.
 *
 * This function is only called once at startup to populate the singleton
 * instance of the kgenv stuct.
 *
 * @param env A pointer to the global environment instance.
 */
void initialize_environment(kgenv* env){

    char* cwd;
    cwd = getcwd(NULL , CWD_BUFFER_SIZE);

    if(cwd == NULL){
        perror("Can't get current working directory\n");
        exit(2);
    }

    env->cwd = cwd;
    env->pwd = NULL;
    env->prompt = "";
    env->uid = getuid();
    env->pword_entry = getpwuid(env->uid);
    env->homedir = env->pword_entry->pw_dir;
    env->path = get_path();
    env->hist = NULL;
    env->aliases = NULL;
    env->watchmails = NULL;
    env->noclobber = true;
}

/**
 * @brief Function executed when SIGINT (Ctrl-C) is caught.
 *
 * @param signal Signal passed in.  Currently it's always SIGINT.
 */
void sig_interrupt(int signal){
    printf("\n");
    // TODO: determine if anything special needs to be done to forward SIGINT to
    // a child process
}
```

```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "redirection.h"

const char* REDIRECT_OPERATORS[] = { ">>&", ">>", ">&", ">", "<" };
const int NUM_REDIRECT_OPERATORS = 5;

/**
 * @brief Parses a command into seperate parts based on redirection operators.
 *
 * @param command The command that will be executed with output/input
 * redirection.
 * @param file The file that needs to opened to properly execute the
 * redirection.  (May be opened for reading or writing.)
 * @param line  The command string input to be parsed.
 *
 * @return Returns an enum value from ::redirect_opcodes indicating the type of
 * redirection that needs to be performed.
 */
enum redirect_opcodes parse_redirection(char** command, char** file,
        char* line){

    char* rd_stdout = NULL;
    enum redirect_opcodes redirect_code;

    for(int i=0; i < NUM_REDIRECT_OPERATORS && rd_stdout == NULL; i++){
        rd_stdout = strstr(line, REDIRECT_OPERATORS[i]);
        redirect_code = i;
    }

    if(rd_stdout == NULL){
        return RD_NONE;
    }

    int command_length = (int)rd_stdout − (int)line;
    *command = (char*)malloc(command_length + 1);
    memcpy(*command, line, command_length);
    (*command)[command_length − 1] = ’\0’;

    int file_length = strlen(line) − (int)rd_stdout + (int)line;
    char* ptr = strtok(line + command_length, " >&<");
    *file = (char*)malloc(file_length);
    memcpy(*file, ptr, strlen(ptr) + 1);

    return redirect_code;
}

/**
 * @brief Performed the redirection actions.
 *
 * @param fid The file id of the redirect file.
 * @param redirect_file The path to the redirect file.
 * @param rt The redirecttion type from ::redirect_opcodes.
 */
void perform_redirection(int* fid, char* redirect_file,
        enum redirect_opcodes rt){

    int open_flags = O_CREAT;

    // Assign read/write mode
    switch(rt){
```

```c
        case RD_ALL_APPEND:
        case RD_STDOUT_APPEND:
        case RD_ALL:
        case RD_STDOUT:
            open_flags |= O_WRONLY;
            break;
        case RD_STDIN:
            open_flags |= O_RDONLY;
            break;
    }

    // Assign append mode
    switch(rt){
        case RD_ALL_APPEND:
        case RD_STDOUT_APPEND:
            open_flags |= O_APPEND;
            break;
        case RD_ALL:
        case RD_STDOUT:
        case RD_STDIN:
    }

    *fid = open(redirect_file, open_flags, 0666);

    // Perform the redirection
    switch(rt){
        case RD_ALL_APPEND:
        case RD_ALL:
            close(2);
            dup(*fid);
            // Fall through (we never redirect only stderr)
        case RD_STDOUT_APPEND:
        case RD_STDOUT:
            close(1);
            dup(*fid);
            close(*fid);
            break;
        case RD_STDIN:
            close(0);
            dup(*fid);
            close(*fid);
    }

}


/**
 * @brief Resets redirection so that stdin, stdout, and stderr all go to the
 * terminal.
 *
 * @param fid Redirection file.
 * @param redirection_type Redirection type from ::redirection_opcodes.
 */
void reset_redirection(int* fid, enum redirect_opcodes redirection_type){
    if(redirection_type != RD_NONE && redirection_type != RD_STDIN){
        *fid = open("/dev/tty", O_WRONLY);
        close(2);
        dup(*fid);
        close(*fid);

        *fid = open("/dev/tty", O_WRONLY);
        close(1);
        dup(*fid);
        close(*fid);
    } else if(redirection_type == RD_STDIN){
        *fid = open("/dev/tty", O_RDONLY);
        close(0);
        dup(*fid);
        close(*fid);
    }
}
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file util.c
 * @brief Definitions of utility functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009−08−02
 */
#include "util.h"
#include "alias.h"
#include "builtins.h"
#include "get_path.h"
#include "wildcard.h"
#include "redirection.h"
#include "ipc.h"

#include <stdio.h>
#include <unistd.h>
#include <readline/history.h>

/**
 * @brief Returns the location of an executable in the PATH.
 *
 * Loops through the path linked list and returns the location of the first
 * file named command in the path directories with execute permissions.  Does
 * not go into sub−directories in search of an executable.
 *
 * @param command Name of the executable to search for.
 * @param pathlist The path list to search.  Usually the one stored in the
 * global ::kgenv environment structure.
 *
 * @return
 */
char* which(const char* command, pathList* pathlist){
    pathList* pl = pathlist;

    // Loop to iterate over every directory in the path
    while(pl != NULL){

        DIR* dirp = opendir(pl−>element);

        if(dirp == NULL){

            perror("Error in which");
            return NULL;

        } else {

            struct dirent* dp = readdir(dirp);        //TODO: errno check?

            // Iterate over every file in the directory
            while(dp != NULL){

                // Determine if the filename matches the command
                if(strcmp(dp−>d_name, command) == 0){
                    closedir(dirp);

                    // Generate an absolute path for the file that was found
                    char* full_path = malloc(strlen(command)
                            + strlen(pl−>element) + 2);
                    if(full_path == NULL){
                        perror("Error in which");
```

```
                        return NULL;
                    }

                    sprintf(full_path, "%s/%s", pl−>element, command);

                    // Check for execute permissions on the file found
                    if(access(full_path, X_OK) == 0){
                        return full_path;
                    } else {
                        //TODO: Verify we don’t need a perror here.  This should
                        //be silent if an error condition occurs.

                        // Free the memory if we’re not returning it
                        free(full_path);
                    }
                }
                dp = readdir(dirp);
            }
            if(closedir(dirp) == −1){
                perror("Error in which");
            }
        }
        pl = pl−>next;
    }
    return NULL;
}


/**
 * @brief Adds a command to the history list.
 *
 * @param command The command to be added to the list.
 * @param env The global ::kgenv environment object.  Needed to access the
 * global history list.
 */
void add_to_history(char* command, kgenv* env){

    //## Add the command to readline’s history
    add_history(command);

    //## Add the command to kgsh history
    histList* new_item;
    new_item = malloc(sizeof(histList));
    if(new_item == NULL){
        perror("Error adding to history");
        return;
    }

    new_item−>command = (char*)malloc(strlen(command) + 1);
    if(new_item−>command == NULL){
        perror("Error adding to history");
        return;
    }

    strcpy(new_item−>command, command);
    new_item−>next = env−>hist;

    if(env−>hist != NULL){
        new_item−>num = env−>hist−>num + 1;
    } else {
        new_item−>num = 1;
    }

    env−>hist = new_item;
}


/**
 * @brief Executes a command.
 *
 * Forks the shell process and executes the given command in the child process.
 * Passes all environment variables.
 *
 * @param cmd The command to be exectued.
 * @param argv Argument array for the command.
 * @param background True if the job needs to be backgrounded
 *
 * @return The exit status of the command.
 */
int exec_cmd(char* cmd, char** argv, bool background){

    //TODO: Print absolute path even if relative is passed?
    #ifdef O_VERBOSE_EXE

    // Print out what’s being executed and if it is backgrounded
    printf("Executing %s%s\n", cmd, (background ? " in background":""));
    fflush(stdout);
    #endif //O_VERBOSE_EXE

    pid_t child_pid = fork();
    int child_status;

    if(child_pid == 0){                          //** Executed in child process

        execve(cmd, argv, environ);

        // Exec commands only return if there’s an error
        perror("Error in exec");

        // We exit since the process image will be replaced with itself here and
        // we will need to enter "exit" twice to truely exit.
        exit(0);

    } else if(child_pid > 0) {                   //** Executed in parent process

        if(!background){

            // If the job isn’t backgrounded, wait for child process to return
            if(!waitpid(child_pid, &child_status, 0)){
                perror("Error in waidpid");
            }

        } else {

            // TODO: add signal handler for SIGCHLD
            if(waitpid(child_pid, &child_status, WNOHANG | WNOWAIT) == −1){
                perror("Error in backgrounding waitpid");
            }

        }

        // Print out the exit status if it is non−zero
        if(WEXITSTATUS(child_status) != 0){
            printf("Exit %d\n", WEXITSTATUS(child_status));
        }

    } else {                                     //** Didn’t fork properly

        perror("Fork failed\n");

    }

    return child_status;
}


/**
 * @brief Processes an input command line.
```

```
 *
 * Processes an input command line entered at the shell prompt from tokenizing
 * through execution.  Handles wildcards, aliases, built−in commands, relative
 * and absolute paths, and any other command line syntax.
 *
 * This function is called primarily through the closed prompt loop in ::main.
 * Memory allocation and deallocation of line_in is handled by ::main.
 *
 * @param line_in The command line entered at the shell prompt.
 * @param global_env The ::kgenv global environment structure.
 * @param deref_alias True if being called on an expanded alias.  False
 * otherwise.  Needed to allow aliases to override commands without causing
 * circular references.
 *
 * @return The length of the line processed.
 */
int process_command_in(char* line_in, kgenv* global_env, bool deref_alias,
        bool blocking){

    int    in_argc;             // argc for the command being processed
    char** in_argv;             // argv for the command being processed
    int    line_length;         // The length of the input line
    bool   background = false;  // True if the command needs to be backgrounded
    int fid;

    line_length = strlen(line_in);
    if(line_in[line_length − 1] == ’\n’)      // Remove trailing newline
        line_in[line_length −1] = ’\0’;

    //## Capture an EOF with no prefix
    if(feof(stdin)){
        //printf("\nUse \"exit\" to leave kgsh.\n");
        //TODO: Fix this feature.
    }

    //## Add the line to the history stack
    if(line_in[0] != ’\0’        // don’t add blank lines
            && !deref_alias){    // don’t add the second call for an alias
        add_to_history(line_in, global_env);
    }


    //## Expand wildcards
    if(contains_wildcards(line_in)){
        char* line_in_original = line_in;
        line_in = expand_wildcards(line_in);
        free(line_in_original);
    }

    //## Process redirection operators
    char* command_line = NULL;
    char* redirect_file = NULL;
    int redirection_type = parse_redirection(&command_line, &redirect_file,
            line_in);

    //TODO: free redirect_file
    if(redirection_type >= 0){
        // Appending or clobbering, or file doesn’t exist
        if(!global_env−>noclobber || redirection_type == RD_ALL_APPEND ||
                redirection_type == RD_STDOUT_APPEND ||
                redirection_type == RD_STDIN ||
                access(redirect_file, F_OK) == −1 ){

            perform_redirection(&fid, redirect_file, redirection_type);

        // File exists and not appending or clobbering
        } else {
            printf("File %s exists.  Overwrite? (y/n) ", redirect_file);
            char c = getchar();
            getchar();
            if(c == 'y' || c == 'Y'){
                if(remove(redirect_file) == -1){
                    perror("Error removing existing redirect file");
                }
                perform_redirection(&fid, redirect_file, redirection_type);
            } else {
                return line_length;
            }
        }

        // Remove the redirection part of the command before continuing
        char* line_in_original = line_in;
        line_in = command_line;
        free(line_in_original);
    }

    //## Process IPC
    if(contains_ipc(line_in)){
        char *left, *right;
        enum ipc_opcodes ipc_type;
        ipc_type = parse_ipc_line(&left, &right, line_in);

        //printf("Piping '%s' to '%s'\n", left, right);
        perform_ipc(left, right, ipc_type, global_env);

        free(left);
        free(right);
        return 0;

    }

    //## Tokenize the line
    //TODO: free in_argv
    in_argv = (char**)calloc(MAX_TOKENS_PER_LINE, sizeof(char*));
    if(in_argv == NULL){
        perror("Error processing command");
        return 0;
    }

    if(!parse_line(&in_argc, &in_argv, &background, line_in)){
        reset_redirection(&fid, redirection_type);
        free(in_argv);
        free(line_in);
        return line_length;        // continue if the line is blank
    }

    //## Check for aliases (Do before builtins to allow for aliasing
    //## builtin commands.
    if(!deref_alias){
        aliasList* alias_ptr = is_alias(global_env, in_argv[0]);
        if(alias_ptr){
            char* new_line_in = (char*)malloc(strlen(alias_ptr->string) + 1);
            strcpy(new_line_in, alias_ptr->string);

            int length = process_command_in(new_line_in, global_env, true,
                    blocking);
            detokenize(alias_ptr->string, length);

            reset_redirection(&fid, redirection_type);
            free(in_argv);
            free(line_in);
            return line_length;
        }
    }


    //## Process built in commands
    int builtin_code = is_builtin(in_argv[0]);
```

```
    if(builtin_code){
        #ifdef O_VERBOSE_EXE
        printf("Executing builtin %s\n", in_argv[0]);
        #endif //O_VERBOSE_EXE
        (*BUILT_IN_FUNCS[--builtin_code])(global_env, in_argc, in_argv);

        reset_redirection(&fid, redirection_type);
        free(in_argv);
        free(line_in);
        return line_length;
    }

    //## Process absolute and relative paths
    // TODO: cleanup this logic
    if( (in_argv[0][0] == '/') ||
        ((in_argv[0][0] == '.') && ((in_argv[0][1] == '/') ||
             (in_argv[0][1] == '.') && (in_argv[0][2] == '/')))){

        // Execute the file if it's executable
        if(access(in_argv[0], X_OK) == 0){
            exec_cmd(in_argv[0], in_argv, background || !blocking);

            reset_redirection(&fid, redirection_type);
            free(in_argv);
            free(line_in);
            return line_length;
        }
    }


    //## Process commands in the path
    char* exe_path = which(in_argv[0], global_env->path);
    if(exe_path != NULL){

        exec_cmd(exe_path, in_argv, background || !blocking);

        reset_redirection(&fid, redirection_type);
        free(in_argv);
        free(line_in);
        free(exe_path);
        return line_length;

    }

    //## Command not found
    fprintf(stderr, "%s: Command not found.\n", in_argv[0]);

    reset_redirection(&fid, redirection_type);
    free(in_argv);
    free(line_in);
    return line_length;
}

/**
 * @brief Parses a command line into an argument (argv) array.
 *
 * @param argc Will be set to the number of arguments in the command string.
 * @param argv Will be set to point to the array of arguments in the command
 * string.  This argument should be preallocated to be an array of pointers.
 * The returned array will point to memory locations inside of line, so it's
 * important that line is not deleted before appropriate action is taken.
 * @param background Will be set to true if the job needs to be backgrounded
 * (i.e. if an & is the last character on the line).
 * @param line The input line to parse.
 *
 * @return 1 if the command was successfully parsed, and 0 if the line is blank.
 */
int parse_line(int* argc, char*** argv, bool* background, char* line){
    int line_length = strlen(line);
```

```c
    //## Check if job needs to be backgrounded
    if(line[line_length − 1] == ’&’){
        line[line_length − 1] = ’\0’;        // Remove the ’&’ character
        *background = true;
    } else {
        *background = false;
    }

    //## Tokenize the command into the argv array
    char* strtok_ptr = NULL;
    char* token = strtok_r(line, " \n", &strtok_ptr);


    // If the line is blank, the first token will be the null string.
    if(token == ’\0’)
        return 0;

    *argv[0] = token;        // argv[0] is the command name

    for(int i = 1; token != NULL && i < MAX_TOKENS_PER_LINE; i++){
        token = strtok_r(NULL, " \t", &strtok_ptr);
        (*argv)[i] = token;
        *argc = i;
    }

    return 1;
}

/**
 * @brief Detokenizes a string that was tokenized using ::strtok.
 *
 * Used primarily by alias functions to detokenize the alias string before
 * storing it in the alias linked list.  For this function to work, all tokens
 * must still be stored sequentially in memory as they are after a call to
 * ::strtok.
 *
 * @param str Pointer to the start of the string.
 * @param length The length of the string in characters.
 */
void detokenize(char* str, int length){
    for(int i=0; i < length − 1; i++){
        if(str[i] == ’\0’){
            str[i] = ’ ’;
        }
    }
}

/**
 * @brief Sets an environment variable.
 *
 * Sets an environment variable in the ::kgenv global environment structure’s
 * internal environment string.  Special action is taken if either the HOME or
 * PATH environment variables change since other data structres need to be
 * updated.
 *
 * @param env The global ::kgenv environment structure.
 * @param name The name of the environment variable to set.
 * @param value The value (string) to set the environment variable to.
 */
void set_environment(kgenv* env, char* name, char* value){

    // Store the new environment variable
    char* str = malloc(strlen(name) + strlen(value) + 2);
    sprintf(str, "%s=%s", name, value);
    putenv(str);

    // Handle a change to HOME
    if(strcmp(name, "HOME") == 0){
```

```c
        //TODO: improve?
        env−>homedir = str + 5;
    }

    // Handle a change to PATH
    else if(strcmp(name, "PATH") == 0){
        //TODO: check for memory leaks here
        pathList* p = env−>path;
        pathList* old;

        // Only free the first one since they are malloced together.
        free(p−>element);
        while(p != NULL){
            old = p;
            p = p−>next;
            free(old);
        }

        env−>path = get_path();
    }
}
```

```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file watchmail.c
 * @brief Contains functions to provide functionality for watchmail builtin
 * @author Kevin Graney
 * @version v0.1
 * @date 2009−09−29
 */
#include "types.h"
#include "watchmail.h"

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <pthread.h>

/**
* @brief Enables and disables watchmail threads.
*
* @param file The filename to control the thread for.
* @param disable If true disables the thread, and if false creates a new thread.
* @param env The global ::kgenv environment object.
*
* @return 0
*/
int control_watchmail(char* file, bool disable, kgenv* env){

    //TODO: error checking
    if(!disable){



        // Create a new node in the watchmails linked list
        watchmailList* new_node = (watchmailList*)malloc(sizeof(watchmailList));
        new_node−>filename = file;
        new_node−>next = env−>watchmails;
        env−>watchmails = new_node;

        // Spawn a thread to monitor the new file
        pthread_create(&(new_node−>thread), NULL, watchmail_thread,
                (void*)(new_node−>filename));

    } else {

        // Loop through watchmail linked list to find the pthread_t record
        watchmailList* prev = NULL;
        watchmailList* curr = env−>watchmails;

        while(curr != NULL){
            if(!strcmp(curr−>filename, file)){
                break;
            }

            prev = curr;
            curr = curr−>next;
        }

        // Case where a thread doesn’t exist for the file
        if(curr == NULL){
            fprintf(stderr, "No watchmail thread for %s exists!\n", file);
            return −1;
```

```c
        }

        pthread_cancel(curr−>thread);

        // Delete the node from the watchmails linked list
        if(prev != NULL){
            prev−>next = curr−>next;
        } else {
            env−>watchmails = curr−>next;
        }

        // Free the memory allocated for the node in the linked list
        free(curr−>filename);
        free(curr);

    }


    return 0;
}


/**
* @brief The pthread function executing for each watchmail thread.
*
* @param param A pointer to a char[] containing the filename to watch for new
* mail in
*
* @return NULL
*/
void* watchmail_thread(void* param){

    char* filename = (char*)param;

    struct stat stat_info;
    off_t last_size;          ///< The file size from the last iteration

    stat(filename, &stat_info);
    last_size = stat_info.st_size;

    // Closed loop to monitor file
    while(1) {
        stat(filename, &stat_info);

        if(stat_info.st_size > last_size){

            struct timeval tp;
            gettimeofday(&tp, NULL);

            printf("\n\aYou have new mail in %s at %s\n", filename,
                    ctime(&(tp.tv_sec)));

        }

        last_size = stat_info.st_size;
        sleep(1);
    }

    return NULL;
}
```

```c
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file wildcard.c
 * @brief Definitions of wildcard functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009−08−02
 */
#include "util.h"
#include "wildcard.h"


/**
 * @brief Determines if an input string contains a wildcard.
 *
 * Used in process_command_in() to determine if wildcard processing can be
 * bypassed or not.  Wildcards checked for are defined in ::WILDCARD_CHARS.
 *
 * @param line The input command line to check for a wildcard in.
 *
 * @return True if a wildcard is present.  False otherwise.
 */
bool contains_wildcards(char* line){

    //## Search the line for any of the wildcard characters.  Return true at the
    //## first match.
    for(int i=0; WILDCARD_CHARS[i] != '\0'; i++){
        if(strchr(line, WILDCARD_CHARS[i]) != NULL){
            return true;
        }
    }

    return false;       // We didn't find any wildcards.
}


/**
 * @brief Expands the wildcards present in an input string.
 *
 * Expands all the wildcards present in the input string based on the current
 * working directory.  First the line is parsed into an argv array and each
 * argument is expanded individually using glob(3C).  Next the expanded
 * arguments are combined back in order to form a single expanded string.
 *
 * \note The return value from this function is a pointer to the heap.  The
 * returned pointer should be freed when not needed anymore.
 *
 * @param line The line to expand.
 *
 * @return The expanded version of line.
 */
char* expand_wildcards(char* line){

    //## Parse the line into arguments
    int    argc;
    char** argv;
    bool   background;

    argv = (char**)calloc(MAX_TOKENS_PER_LINE, sizeof(char*));
    if(argv == NULL){
        perror("Error while expanding wilcards");
        return NULL;
```

```c
    }
    parse_line(&argc, &argv, &background, line);

    //## Expand all the arguments individually
    char** expanded_argv = (char**)calloc(argc, sizeof(char*));
    if(expanded_argv == NULL){
        perror("Error while expanding wilcards");
        return NULL;
    }

    for(int i=0; i < argc; i++){
        expanded_argv[i] = expand_argument(argv[i]);
    }

    //## Find the total length the expanded line will be
    int length = 0;
    for(int i=0; i < argc; i++){
        length += strlen(expanded_argv[i]) + 1;
    }

    //## Form expanded line by concatenating all the expanded arguments
    char* expanded = calloc(length, sizeof(char));
    if(expanded == NULL){
        perror("Error while expanding wilcards");
        return NULL;
    }

    for(int i=0; i < argc; i++){
        strcat(expanded, " ");
        strcat(expanded, expanded_argv[i]);
        free(expanded_argv[i]);
    }

    //## Free up memory
    free(expanded_argv);
    free(argv);

    return expanded;
}



/**
 * @brief Expands wildcards in a single argument string.
 *
 * Called by expand_wildcards(), this function expands a single argument in the
 * argv array by calling glob(3C).
 *
 * \note This function returns a pointer to the heap, therefore the pointer must
 * be freed after use.  A copy of the input argument is returned if no wildcards
 * are present to prevent issues with deallocating memory that is allocated
 * outside this function.
 *
 * @param argument The argument string to be expanded.
 *
 * @return The expanded argument.  If no wildcards are present in the string a
 * copy of the argument parameter is returned.
 */
char* expand_argument(char* argument){

    glob_t pglob;

    if(glob(argument, 0, NULL, &pglob) == 0){   //TODO: errno handling

        //## If no wildcard in the argument return a copy of itself
        if(pglob.gl_pathc == 0){
            char* argument_copy = malloc(strlen(argument) + 1);
            if(argument_copy == NULL){
                perror("Error expanding argument");
                globfree(&pglob);
                return NULL;
            }

            strcpy(argument_copy, argument);
            globfree(&pglob);           // Free up memory
            return argument_copy;
        }

        //## Determine total length of expanded argument
        int length = 0;
        for(int i=0; i < pglob.gl_pathc; i++){
            length += strlen(pglob.gl_pathv[i]) + 1;
        }

        //## Allocate new space for the expanded argument
        char* expanded_arg = calloc(length, sizeof(char));
        if(expanded_arg == NULL){
            perror("Error expanding argument");
            globfree(&pglob);
            return NULL;
        }

        //## Form expanded argument string
        for(int i=0; i < pglob.gl_pathc; i++){
            strcat(expanded_arg, " ");
            strcat(expanded_arg, pglob.gl_pathv[i]);
        }

        //## Free up memory
        globfree(&pglob);

        return expanded_arg;

    } else {

        //## Make a copy of the argument and return
        char* argument_copy = malloc(strlen(argument) + 1);
        if(argument_copy == NULL){
            perror("Error expanding argument");
            globfree(&pglob);
            return NULL;
        }

        //## Free up memory
        globfree(&pglob);

        strcpy(argument_copy, argument);
        return argument_copy;
    }
}
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file alias.h
 * @brief Definition of alias functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-08-02
 */
#ifndef _ALIAS_INC
#define _ALIAS_INC

#include <errno.h>
#include "types.h"

void add_alias(kgenv* env, char* name, int cmd_argc, char* cmd_argv[]);

bool remove_alias(kgenv* env, char* name);

aliasList* is_alias(kgenv* env, char* name);

#endif //_ALIAS_INC
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file builtins.h
 * @brief Declarations of builtin functions and constant members.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-08-02
 */
#ifndef _BUILTINS_INC
#define _BUILTINS_INC

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>      // for perror

#include <sys/types.h>  // for readdir and opendir
#include <dirent.h>     // for readdir and opendir

#include <signal.h>     // for sigsend

#include "util.h"
#include "types.h"

//-- Constants to define how many builtins we have
#ifdef DEBUG
#define NUM_BUILTINS 25         // Total number of commands, including debug
#else
#define NUM_BUILTINS 19         // Total number of commands, excluding debug
#endif //DEBUG

short int is_builtin(char* command);

void (*BUILT_IN_FUNCS[NUM_BUILTINS])(kgenv*, int, char**);

//-----------------------------------------------------------------------------
//-- The following are functions that define each built in command.  Each
//-- function takes a pointer to the shell's environment structure followed by
//-- an argc and argv passed on from the shell. Functions are named such that a
//-- function bic_foo is run when the foo command is issued.
//-----------------------------------------------------------------------------
void bic_exit(kgenv* env, int argc, char* argv[]);

void bic_which(kgenv* env, int argc, char* argv[]);

void bic_where(kgenv* env, int argc, char* argv[]);

void bic_cd(kgenv* env, int argc, char* argv[]);

void bic_pwd(kgenv* env, int argc, char* argv[]);

void bic_list(kgenv* env, int argc, char* argv[]);

void bic_pid(kgenv* env, int argc, char* argv[]);

void bic_kill(kgenv* env, int argc, char* argv[]);

void bic_prompt(kgenv* env, int argc, char* argv[]);

void bic_printenv(kgenv* env, int argc, char* argv[]);

void bic_alias(kgenv* env, int argc, char* argv[]);
```

```
void bic_unalias(kgenv* env, int argc, char* argv[]);

void bic_history(kgenv* env, int argc, char* argv[]);

void bic_setenv(kgenv* env, int argc, char* argv[]);

void bic_lsbuiltins(kgenv* env, int argc, char* argv[]);

void bic_watchmail(kgenv* env, int argc, char* argv[]);

void bic_noclobber(kgenv* env, int argc, char* argv[]);

void bic_vimode(kgenv* env, int argc, char* argv[]);

void bic_emacsmode(kgenv* env, int argc, char* argv[]);

//-----------------------------------------------------------------------------
//-- The following are functions associated with debugging commands and are
//-- intended for development use only.  Compile with -DDEBUG for use.
//-----------------------------------------------------------------------------
#ifdef DEBUG
void _db_tokenizer(kgenv* env, int argc, char* argv[]);

void _db_kgenv(kgenv* env, int argc, char* argv[]);

void _db_path(kgenv* env, int argc, char* argv[]);

void _db_history(kgenv* env, int argc, char* argv[]);

void _db_wc_contains(kgenv* env, int argc, char* argv[]);

void _db_wc_expand(kgenv* env, int argc, char* argv[]);
#endif //DEBUG

#endif //_BUILTINS_INC
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/*
  get_path.h
  Ben Miller

*/
#ifndef _GET_PATH_INC
#define _GET_PATH_INC

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

/* function prototype.  It returns a pointer to a linked list for the path
   elements. */
struct pathelement *get_path();

struct pathelement
{
  char *element;                        /* a dir in the path */
  struct pathelement *next;             /* pointer to next node */
};


#endif //_GET_PATH_INC
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file ipc.c
 * @brief Definitions of IPC functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-10-25
 */
#ifndef _IPC_INC
#define _IPC_INC

#include "types.h"
#include <string.h>

enum ipc_opcodes {
    IPC_ALL,         // The |&  operator
    IPC_STDOUT       // The | operator
};

bool contains_ipc(char* line);

enum ipc_opcodes parse_ipc_line(char** left, char** right, char* line);

void perform_ipc(char* left, char* right, enum ipc_opcodes ipc_type,
        kgenv* global_env);

#endif //_IPC_INC
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

#ifndef REDIRECTION_H
#define REDIRECTION_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

enum redirect_opcodes {
    RD_ALL_APPEND,      // The >>& operator
    RD_STDOUT_APPEND,   // The >>  operator
    RD_ALL,             // The >&  operator
    RD_STDOUT,          // The >   operator
    RD_STDIN,           // The <   operator
    RD_NONE = −1        // No redirect operator
};

enum redirect_opcodes parse_redirection(char** command, char** file,
        char* line);

void perform_redirection(int* fid, char* redirect_file,
        enum redirect_opcodes redirection_type);

void reset_redirection(int* fid, enum redirect_opcodes redirection_type);

#endif //REDIRECTION_H
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file types.h
 * @brief Type declarations.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-08-02
 */
#ifndef _TYPES_INC
#define _TYPES_INC

#include "get_path.h"

#define false 0     ///< C++ style false keyword
#define true  1     ///< C++ style true keyword
typedef char bool;  ///< Fake boolean in C++ style


//-----------------------------------------------------------------------------
//-- Linked List Types
//-----------------------------------------------------------------------------


/**
 * @brief Struct to represent each node in the history linked list.
 */
struct histelement {
    int    num;                 ///< Command number
    char*  command;             ///< Command string
    struct histelement* next;   ///< Pointer to next node (previous command)
};

/**
 * @brief Typedef to refer to the history linked list.
 */
typedef struct histelement histList;

/**
 * @brief Structure to represent each node in the aliases linked list.  Take
 * note that the commands are stored in their unparsed condition to make
 * the code cleaner.
 */
struct aliaselement {
    char*  name;                ///< The name of the alias
    char*  string;              ///< Command string alias refers to
    struct aliaselement* next;  ///< Pointer to next node
};

/**
 * @brief Typedef to refer to the alias linked list.
 */
typedef struct aliaselement aliasList;

/**
 * @brief List of files and pthread_t structures that are currently being
 * watched by the ::watchmail builtin.
 */
struct watchmailelement {
    char* filename;             ///< Path to the file being watched
    pthread_t thread;           ///< ::thread_t structure for ::watchmail_thread
    struct watchmailelement* next; ///< Pointer to next node
};
```

```
/**
 * @brief Typedef to refer to the watchmail linked list.
 */

typedef struct watchmailelement watchmailList;
/**
 * @brief A typedef is defined for the ::pathelement struct to be consistent
 * with the other linked lists.
 */
typedef struct pathelement pathList;


//-----------------------------------------------------------------------------
//-- Environment Types
//-----------------------------------------------------------------------------


/**
 * @brief Global environment structure.
 *
 * The kgenv type will contain our current environment.  If this were being done
 * in C++ it would be a singleton class since we only ever create one variable
 * of this type.  Basically, we're encapsulating all our would be global
 * variables into a nice neat structure.
 */
typedef struct {
    int uid;                            ///< User ID
    char* homedir;                      ///< Home directory path
    struct passwd *pword_entry;         ///< Passwd entry info (not needed?)

    char* cwd;                          ///< Current working directory
    char* pwd;                          ///< Prior working directory

    bool  noclobber;            ///< Clobber variable for file redirection

    char* prompt;                       ///< Prompt prefix string
    pathList* path;                     ///< Path list pointer
    histList* hist;                     ///< History list pointer
    aliasList* aliases;                 ///< Alias list pointer
    watchmailList* watchmails;  ///< Watchmail list pointer
} kgenv;


//-----------------------------------------------------------------------------
//-- Function Types
//-----------------------------------------------------------------------------

/**
 * @brief This is the generic function type for a built in function.  It's used
 * to setup the function pointer arrays.
 */
typedef void (*bicfunc)(kgenv*, int, char*);

#endif //_TYPES_INC
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file util.h
 * @brief Definitions of utility functions.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-08-02
 */
#ifndef _UTIL_INC
#define _UTIL_INC

#include <stdio.h>
#include <stdlib.h>

#include <sys/wait.h>   // for waitpid and WEXITSTATUS
#include <sys/types.h>  // for readdir and opendir
#include <dirent.h>     // for readdir and opendir

#include <string.h>

#include "types.h"
#include "get_path.h"


//#define  O_VERBOSE_EXE        // Enable the "Executing ..." messages

#define CWD_BUFFER_SIZE         1024
#define LINE_BUFFER_SIZE        1024
#define MAX_TOKENS_PER_LINE 512
#define HISTORY_SIZE            1024


/**
 * @brief The external environment variable list from the calling shell.
 */
extern char** environ;

char* which(const char *command, pathList* pathlist);

void add_to_history(char* command, kgenv* env);

int exec_cmd(char* cmd, char** argv, bool background);

int process_command_in(char* line_in, kgenv* global_env, bool deref_alias,
        bool blocking);

int parse_line(int* argc, char*** argv, bool* background, char* line);

void detokenize(char* str, int length);

void set_environment(kgenv* env, char* name, char* value);

#endif //_UTIL_INC
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file watchmail.h
 * @brief Contains prototypes for watchmail builtin functions
 * @author Kevin Graney
 * @version v0.1
 * @date 2009−09−29
 */
#ifndef _WATCHMAIL_INC
#define _WATCHMAIL_INC

int control_watchmail(char* file, bool disable, kgenv* env);

void* watchmail_thread(void*);

#endif //_BUILTINS_INC
```

```
/*
 * CISC361: Operating Systems (Fall 2009)
 * Instructor: Ben Miller
 *
 * Project 2
 * Kevin Graney
 */

/**
 * @file wildcard.h
 * @brief Declarations of wildcard functions and constant members.
 * @author Kevin Graney
 * @version v0.1
 * @date 2009-08-02
 */
#ifndef _WILDCARD_H
#define _WILDCARD_H

#include <string.h>
#include <glob.h>
#include <errno.h>
#include "types.h"

#define MAX_WILDCARDS           512
#define WILDCARD_CHARS          "*?"

bool contains_wildcards(char* line);

char* expand_wildcards(char* line);

char* expand_argument(char* argument);

#endif //_WILDCARD_H
```