```
// ACADEMIC INTEGRITY PLEDGE
//
// - I have not used source code obtained from another student nor
//   any other unauthorized source, either modified or unmodified.
//
// - All source code and documentation used in my program is either
//   my original work or was derived by me from the source code
//   published in the textbook for this course or presented in
//   class.
//
// - I have not discussed coding details about this project with
//   anyone other than my instructor. I understand that I may discuss
//   the concepts of this program with other students and that another
//   student may help me debug my program so long as neither of us
//   writes anything during the discussion or modifies any computer
//   file during the discussion.
//
// - I have violated neither the spirit nor letter of these restrictions.
//
//
//
// Signed:_____ Date:_____

// 3460:426 Lab 1 - Basic C shell rev. 9/10/2020

/* Basic shell */

/*
 * This is a very minimal shell. It finds an executable in the
 * PATH, then loads it and executes it (using execv). Since
 * it uses "." (dot) as a separator, it cannot handle file
 * names like "minishell.h"
 *
 * The focus on this exercise is to use fork, PATH variables,
 * and execv.
 */

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

#define MAX_ARGS                64
#define MAX_ARG_LEN             16
#define MAX_LINE_LEN    80
#define WHITESPACE              " ,\t\n"

struct command_t {
   char *name;
   int argc;
   char *argv[MAX_ARGS];
};

/* Function prototypes */
int parseCommand(char *, struct command_t *);
void printPrompt();
void readCommand(char *);

int main(int argc, char *argv[]) {
   int pid;
   int status;
   char cmdLine[MAX_LINE_LEN];
   struct command_t command;

   while (TRUE) {
      printPrompt();
      /* Read the command line and parse it */
      readCommand(cmdLine);
      parseCommand(cmdLine, &command);
      command.argv[command.argc] = NULL;

          /*
             TODO: if the command is one of the shortcuts you're testing for
                 either execute it directly or build a new command structure to
                 execute next
           */

      /* Create a child process to execute the command */
      if ((pid = fork()) == 0) {
         /* Child executing command */
         execvp(command.name, command.argv);
         /* TODO: what happens if you enter an incorrect command? */
      }
      /* Wait for the child to terminate */
      wait(&status); /* EDIT THIS LINE */
   }

   /* Shell termination */
   printf("\n\n shell: Terminating successfully\n");
   return 0;
}

/* End basic shell */

/* Parse Command function */

/* Determine command name and construct the parameter list.
 * This function will build argv[] and set the argc value.
 * argc is the number of "tokens" or words on the command line
 * argv[] is an array of strings (pointers to char *). The last
 * element in argv[] must be NULL. As we scan the command line
 * from the left, the first token goes in argv[0], the second in
 * argv[1], and so on. Each time we add a token to argv[],
 * we increment argc.
 */
int parseCommand(char *cLine, struct command_t *cmd) {
   int argc;
   char **clPtr;
   /* Initialization */
   clPtr = &cLine;      /* cLine is the command line */
   argc = 0;
   cmd->argv[argc] = (char *) malloc(MAX_ARG_LEN);
   /* Fill argv[] */
   while ((cmd->argv[argc] = strsep(clPtr, WHITESPACE)) != NULL) {
      cmd->argv[++argc] = (char *) malloc(MAX_ARG_LEN);
   }

   /* Set the command name and argc */
   cmd->argc = argc-1;
   cmd->name = (char *) malloc(sizeof(cmd->argv[0]));
   strcpy(cmd->name, cmd->argv[0]);
   return 1;
}

/* End parseCommand function */

/* Print prompt and read command functions - Nutt pp. 79-80 */

void printPrompt() {
   /* Build the prompt string to have the machine name,
    * current directory, or other desired information
    */
   promptString = ...; /* EDIT THIS LINE */
   printf("%s ", promptString);
}

void readCommand(char *buffer) {
   /* This code uses any set of I/O functions, such as those in
    * the stdio library to read the entire command line into
    * the buffer. This implementation is greatly simplified,
    * but it does the job.
    */
   fgets(buffer, 80, stdin);
}

/* End printPrompt and readCommand */
```