builtin.c

```
#include "builtin.h"
#include "functions.h"

// returns true if the 'exit' call
// should be performed
int exit_shell(char* cmd) {
    return exitNicely(cmd);  // Your code here
}

// returns true if "chdir" was performed
// this means that if 'cmd' contains:
//  $ cd directory (change to 'directory')
//  $ cd (change to HOME)
// it has to be executed and then return true
int cd(char* cmd) {
    return changeDirectory(cmd);  // Your code here
}

// returns true if 'pwd' was invoked
// in the command line
int pwd(char* cmd) {
    return printWorkingDirectory(cmd);  // Your code here
}
```

builtin.h

```
#ifndef BUILTIN_H
#define BUILTIN_H

#include "defs.h"

extern char promt[PRMTLEN];

int cd(char* cmd);

int exit_shell(char* cmd);

int pwd(char* cmd);

#endif // BUILTIN_H
```

createcmd.c

```
#include "createcmd.h"

// creates an execcmd struct to store
// the args and environ vars of the command
struct cmd* exec_cmd_create(char* buf_cmd) {

    struct execcmd* e;

    e = (struct execcmd*)calloc(sizeof(*e), sizeof(*e));

    e->type = EXEC;
    strcpy(e->scmd, buf_cmd);

    return (struct cmd*)e;
}

// creates a backcmd struct to store the
// background command to be executed
struct cmd* back_cmd_create(struct cmd* c) {

    struct backcmd* b;

    b = (struct backcmd*)calloc(sizeof(*b), sizeof(*b));

    b->type = BACK;
    strcpy(b->scmd, c->scmd);
    b->c = c;

    return (struct cmd*)b;
}

// encapsulates two commands into one pipe struct
struct cmd* pipe_cmd_create(struct cmd** cmdVec, size_t size) {

    if (size == 1) {
        struct cmd* cmd = cmdVec[0];
        free(cmdVec);
        return cmd;
    }
```

```
    struct pipecmd* p;

    p = (struct pipecmd*)calloc(sizeof(*p), sizeof(*p));

    p->type = PIPE;
    p->cmdVec = cmdVec;
    p->size = size;

    return (struct cmd*)p;
}
```

createcmd.h

```
#ifndef CREATECMD_H
#define CREATECMD_H

#include "defs.h"
#include "types.h"

struct cmd* exec_cmd_create(char* cmd);

struct cmd* back_cmd_create(struct cmd* c);

struct cmd* pipe_cmd_create(struct cmd** cmdVec, size_t size);

#endif // CREATECMD_H
```

defs.h

```
#ifndef DEFS_H
#define DEFS_H

#define _GNU_SOURCE

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>

// color scape strings
#define COLOR_BLUE "\x1b[34m"
#define COLOR_RED "\x1b[31m"
#define COLOR_RESET "\x1b[0m"

#define END_STRING '\0'
#define END_LINE '\n'
#define SPACE ' '

#define BUFLEN 1024
#define PRMTLEN 1024
#define MAXARGS 20
#define ARGSIZE 1024
#define FNAMESIZE 200

// Command representation after parsed
#define EXEC 1
#define BACK 2
#define REDIR 3
#define PIPE 4

// Fd for pipes
#define READ 0
#define WRITE 1

#define EXIT_SHELL 1

#endif //DEFS_H
```

exec.c

```
#include "exec.h"
#include "functions.h"
// executes a command - does not return
//
// Hint:
```

```
// - check how the 'cmd' structs are defined
//  in types.h
void exec_cmd(struct cmd* cmd) {

    switch (cmd->type) {

        case EXEC: {
            // spawns a command
            struct execcmd* execcmd = (struct execcmd*) cmd;
            setEnvironmentVariables(execcmd->eargv, execcmd->eargc);
            execCommand(cmd); // Your code here
            printf("Commands are not yet implemented\n");
            _exit(-1);
            break;
        }

        case BACK: {
            // runs a command in background
            runBackground(cmd);  //Your code here
            printf("Background process are not yet implemented\n");
            _exit(-1);
            break;
        }

        case REDIR: {
            // changes the input/output/stderr flow
            runRedir(cmd); // Your code here
            printf("Redirections are not yet implemented\n");
            _exit(-1);
            break;
        }

        case PIPE: {
            // pipes two commands
            runMultiplePipe(cmd); // Your code here
            printf("Pipes are not yet implemented\n");

            // free the memory allocated
            // for the pipe tree structure
            free_command(parsed_pipe);

            break;
        }
    }
}
```

exec.h

```
#ifndef EXEC_H
#define EXEC_H

#include "defs.h"
#include "types.h"
#include "utils.h"
#include "freecmd.h"

extern struct cmd* parsed_pipe;

void exec_cmd(struct cmd* c);

#endif // EXEC_H
```

freecmd.c

```
#include "freecmd.h"

// frees the memory allocated
// for the tree structure command
void free_command(struct cmd* cmd) {

    int i;
    struct pipecmd* p;
    struct execcmd* e;
    struct backcmd* b;

    if (cmd->type == PIPE) {
        p = (struct pipecmd*)cmd;
        for (size_t i = 0; i < p->size; ++i) free_command(p->cmdVec[i]);
        free(p->cmdVec);
        free(p);
        return;
    }

    if (cmd->type == BACK) {
```

```
        b = (struct backcmd*)cmd;

        free_command(b->c);
        free(b);
        return;
    }

    e = (struct execcmd*)cmd;

    for (i = 0; i < e->argc; i++)
        free(e->argv[i]);

    for (i = 0; i < e->eargc; i++)
        free(e->eargv[i]);

    free(e);
}
```

freecmd.h

```
#ifndef FREECMD_H
#define FREECMD_H

#include "defs.h"
#include "types.h"

void free_command(struct cmd* c);

#endif // FREECMD_H
```

functions.c

```
#include "functions.h"
#include "utils.h"
#include "general.h"
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pwd.h>
```

```
#include "exec.h"
#include "printstatus.h"
#include <ctype.h>
#include <stdlib.h>
#include "parsing.h"
//**************************************************************************
// INICIO FUNCIONES ESTATICAS
//**************************************************************************
//--------------------------------------------------------------------------
// IS EQUAL TO IN N BYTES
//--------------------------------------------------------------------------
static int isEqualToInNBytes(const char* str, const char* cmp, size_t bytes) {
    for (size_t i = 0; i < bytes; ++i) {
        if (str[i] != cmp[i]) return false;
    }
    return true;
}
//--------------------------------------------------------------------------
// FIND FIRS CARACTER AFTER SPACE
//--------------------------------------------------------------------------
static int findFirstCaracterAfterSpace(const char* str, int offset) {
    size_t size = strlen(str);
    for (size_t i = 0; i < size; ++i) {
        if (str[i] == SPACE) continue;
        return i+offset;
    }
    return -1;
}
//--------------------------------------------------------------------------
// GET ENVIRONMENT VALUE
//--------------------------------------------------------------------------
// sets the "value" argument with the value part of
// the "arg" argument and null-terminates it
static void getEnvironmentValue(char* arg, char* value, int idx) {
    int i, j;
    for (i = (idx + 1), j = 0; i < strlen(arg); i++, j++) value[j] = arg[i];
    value[j] = END_STRING;
}
//--------------------------------------------------------------------------
// GET ENVIRONMENT KEY
//--------------------------------------------------------------------------
// sets the "key" argument with the key part of
// the "arg" argument and null-terminates it
static void getEnvironmentKey(char* arg, char* key) {
    int i;
    for (i = 0; arg[i] != '='; i++) key[i] = arg[i];
```

```
    key[i] = END_STRING;
}
//------------------------------------------------------------------------------
// REDIR
//------------------------------------------------------------------------------
static int redir(int oldFd, int newFd) {
    if (newFd == -1 || oldFd == -1) return 0;
    int fd = dup2(oldFd, newFd);
    if (fd == -1) perr("ERROR: dup2(%d, %d) filed", oldFd, newFd);
    return fd;
}
//------------------------------------------------------------------------------
// PWD
//------------------------------------------------------------------------------
char* getWorkingDirectory() {
    char* directory = (char*) malloc(sizeof(char)* PATH_MAX);
    getcwd(directory, PATH_MAX);
    size_t i = 1;
    while (directory == NULL) {
        free(directory);
        directory = (char*) malloc(sizeof(char)* PATH_MAX*i);
        getcwd(directory, PATH_MAX);
        i++;
    }
    return directory;
}
//------------------------------------------------------------------------------
// SAVE COMMAND
//------------------------------------------------------------------------------
static int saveCommand(char** command, char* left) {
    size_t size = strlen(left);
    *command = (char*) malloc(sizeof(char) * (size + 1));
    if (!*command) {
        printf("ERROR: command = malloc() failed");
        return 1;
    }
    strncpy(*command, left, size+1);
    return 0;
}
//------------------------------------------------------------------------------
// RUN PIPE
//------------------------------------------------------------------------------
static void runPipe(struct cmd* left, struct cmd* right, int last) {
    int pipeFd[2];
    if (pipe(pipeFd) == -1) perr("ERROR: pipe() failed in function runPipe()");
    // if (redir(pipeFd[0], pipeFd[1]) == -1) perr("ERROR redir failed");
```

```
    int status = 0;

    pid_t p = fork();
    if (p == -1) perr("ERROR fork failed in function runPipe()");
    if (p == 0) {
        // child process
        close(pipeFd[0]); // Close unused write end
        if (redir(pipeFd[1], STDOUT_FILENO) == -1) perr("ERROR redir failed");
        exec_cmd(left);
    } else {
        // parent process
        waitpid(p, &status, 0);
        close(pipeFd[1]); // Close unused read end
        if (redir(pipeFd[0], STDIN_FILENO) == -1) perr("ERROR redir failed");
        if (last) exec_cmd(right);
    }
}
//*************************************************************************
// FIN FUNCIONES ESTATICAS
//*************************************************************************
//-------------------------------------------------------------------------
// EXEC COMMAND
//-------------------------------------------------------------------------
void execCommand(struct cmd* cmd) {
    struct execcmd* execcmd = (struct execcmd*) cmd;
    if (execcmd->argc == 0) return;
    const char* file = (const char*) execcmd->argv[0];
    if (execvp(file, execcmd->argv) == -1) {
        perr("ERROR function execvp() returned -1");
    }
}
//-------------------------------------------------------------------------
// EXPAND ENVIRONMENT VARIABLES
//-------------------------------------------------------------------------
char* expandEnvironmentVariables(char* arg) {
    if (arg[0] != '$') return arg;
    char* value = getenv(arg+1);
    if (value == NULL) return arg;
    size_t sizeArg = strlen(arg);
    size_t sizeValue = strlen(value);
    if (sizeValue > sizeArg) {
        char* expansion = (char*) malloc(sizeof(char)*(sizeValue + 1));
        free(arg);
        strncpy(expansion, value, sizeValue);
        expansion[sizeValue] = END_STRING;
```

```
        return expansion;
    } else {
        strncpy(arg, value, sizeValue);
        arg[sizeValue] = END_STRING;
        return arg;
    }
}
//---------------------------------------------------------------------------
// CD
//---------------------------------------------------------------------------
int changeDirectory(char* cmd) {
    if (!isEqualToInNBytes(cmd, "cd", 2)) return false;
    int idx = findFirstCaracterAfterSpace(cmd+2, 2);
    int value;
    if (idx == -1) {
        value = chdir("/home");
    } else {
        value = chdir(cmd+idx);
    }
    if (value == -1) perr("ERROR function chdir() failed");
    return true;
}
//---------------------------------------------------------------------------
// EXIT
//---------------------------------------------------------------------------
int exitNicely(char* cmd) {
    if (!isEqualToInNBytes(cmd, "exit", 4)) return false;
    int status = 0;
    exit(status & 0377);
    return true;
}
//---------------------------------------------------------------------------
// PWD
//---------------------------------------------------------------------------
int printWorkingDirectory(char* cmd) {
    if (!isEqualToInNBytes(cmd, "pwd", 3)) return false;
    char* directory = getWorkingDirectory();
    printf("%s\n", directory);
    free(directory);
    return true;
}
//---------------------------------------------------------------------------
// SET ENVIRONMENT VARIABLES
//---------------------------------------------------------------------------
void setEnvironmentVariables(char** eargv, int eargc) {
    for (size_t i = 0; i < eargc; ++i) {
```

```
        int idx = block_contains(eargv[i], '=');
        if (idx == -1) continue;
        size_t size = strlen(eargv[i]);
        char* key = (char*) malloc(sizeof(char) * (idx + 1));
        if (key == NULL) continue;
        char* value = (char*) malloc(sizeof(char) * (size - idx));
        if (key == NULL) {
            free(key);
            continue;
        }
        getEnvironmentKey(eargv[i], key);
        getEnvironmentValue(eargv[i], value, idx);
        if (setenv(key, value, 0) == -1) {
            perr("ERROR: function setenv(%s, %s, 0) returned -1", key, value);
        }
        free(key);
        free(value);
    }
}
//------------------------------------------------------------------------------
// RUN BACKGROUND
//------------------------------------------------------------------------------
void runBackground(struct cmd* cmd) {
    struct backcmd* backcmd = (struct backcmd*) cmd;
    exec_cmd(backcmd->c);
}
//------------------------------------------------------------------------------
// EXEC BACKGROUND
//------------------------------------------------------------------------------
int execBackground(struct cmd* cmd) {
    if (cmd->type != BACK) return false;
    print_back_info(cmd);
    return true;
}
//------------------------------------------------------------------------------
// OPEN REDIR FD
//------------------------------------------------------------------------------
int openRedirFd(char* file) {
    size_t size = strlen(file);
    if (size == 0) return -1;
    if (size == 1 && isdigit(file)) return (int) *file;
    int idx = block_contains(file, '&');
    if (idx >= 0 && size == 2) return atoi(file + idx + 1);
    int append = false;
    if (file[0] == '>') append = true;  // Challenge Parte 2: Flujo estándar
```

```
    int fd;
    int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;  // 0644
    if (append) {
        fd = open(&file[1], O_RDWR | O_CREAT | O_APPEND, mode);
    } else {
        fd = open(file, O_RDWR | O_CREAT, mode);
    }
    if (fd == -1) {
        perr("ERROR: open failed with file %s in function openRedirFd()", file);
    }
    return fd;
}
//------------------------------------------------------------------------------
// RUN REDIR
//------------------------------------------------------------------------------
void runRedir(struct cmd* cmd) {
    struct execcmd* execcmd = (struct execcmd*) cmd;

    int fdIn = openRedirFd(execcmd->in_file);
    int fdOut = openRedirFd(execcmd->out_file);
    int fdErr = openRedirFd(execcmd->err_file);

    int redirIn = redir(fdIn, STDIN_FILENO);
    int redirOut = redir(fdOut, STDOUT_FILENO);
    int redirErr = redir(fdErr, STDERR_FILENO);

    if (redirIn == -1 || redirOut == -1 || redirErr == -1) exitNicely("exit");
    cmd->type = EXEC;
    exec_cmd(cmd);
}
//------------------------------------------------------------------------------
// RUN MULTIPLE PIPE
//------------------------------------------------------------------------------
void runMultiplePipe(struct cmd* cmd) {
    struct pipecmd* pipecmd = (struct pipecmd*) cmd;
    int last = false;
    for (size_t i = 0; i < pipecmd->size-1; ++i) {
        if (i == pipecmd->size-2) last = true;
        runPipe(pipecmd->cmdVec[i], pipecmd->cmdVec[i+1], last);
    }
}
//------------------------------------------------------------------------------
// PARSING
//------------------------------------------------------------------------------
int parsing(struct execcmd* c, char* arg) {
    size_t size = strlen(arg);
```

```
    size_t append = 0;
    size_t redirErrOut = 0;
    for (size_t i = 0; i < size; ++i) {
        if (arg[i] == '>' && (i == 0 || i == 1)) append++;
        if (arg[i] == '&' && i == 0) redirErrOut++;
    }

    if (redirErrOut == 1 && append == 1) {
        strcpy(c->out_file, &arg[2]);
        strcpy(c->err_file, &arg[2]);
        free(arg);
        c->type = REDIR;
        return true;
    }
    return false;
}
//-------------------------------------------------------------------------------
// GET SIZE
//-------------------------------------------------------------------------------
int getSize(char* cmd, char* commands[], int (*f)(char**, char*)) {
    int i = 0;
    char* aux = (char*) malloc(sizeof(char) * (strlen(cmd) + 1));
    strcpy(aux, cmd);
    char* left = aux;
    char* right;
    while (block_contains(left, '|') >= 0) {
        right = split_line(left, '|');
        if (f) f(&commands[i], left);
        left = right;
        i++;
    }
    if (f) f(&commands[i], left);
    free(aux);
    return ++i;
}
//-------------------------------------------------------------------------------
// PRINT COMMAND
//-------------------------------------------------------------------------------
int printCommand(char* commands[], size_t size) {
    printf("{ ");
    for (size_t j = 0; j < size; ++j) {
        if (j == size-1) printf("%s", commands[j]);
        else printf("%s, ", commands[j]);
    }
    printf(" }\n");
    return 0;
```

```
}
//------------------------------------------------------------------------------
// FREE COMMAND
//------------------------------------------------------------------------------
int freeCommand(char* commands[], size_t size) {
    for (size_t j = 0; j < size; ++j) free(commands[j]);
    free(commands);
    return 0;
}
//------------------------------------------------------------------------------
// GET COMMANDS
//------------------------------------------------------------------------------
char** getCommands(char* cmd, size_t* size) {
    *size = getSize(cmd, NULL, NULL);
    char** commands = (char**) malloc(sizeof(char*) * (*size + 1));
    if (!commands) {
        perr("ERROR: commands = malloc() failed");
        return NULL;
    }
    commands[*size] = NULL;
    getSize(cmd, commands, saveCommand);
    return commands;
}
//------------------------------------------------------------------------------
// CREATE COMMANDS
//------------------------------------------------------------------------------
struct cmd* createCommands(char* cmd) {
    size_t size;
    char** commands = getCommands(cmd, &size);
    size_t posSize = sizeof(struct cmd*);
    struct cmd** cmdVec = (struct cmd**) malloc(posSize*(size+1));
    if (!cmdVec) {
        perr("ERROR: cmdVec = malloc() failed");
        freeCommand(commands, size);
        return NULL;
    }
    for (size_t i = 0; i < size; ++i) cmdVec[i] = parse_cmd(commands[i]);
    freeCommand(commands, size);
    cmdVec[size] = NULL;
    return pipe_cmd_create(cmdVec, size);
}
//------------------------------------------------------------------------------
```

functions.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
//------------------------------------------------------------------------------
// INCLUDES
//------------------------------------------------------------------------------
#include "types.h"
//------------------------------------------------------------------------------
// EXEC COMMAND
//------------------------------------------------------------------------------
void execCommand(struct cmd* cmd);
//------------------------------------------------------------------------------
// EXPAND ENVIRONMENT VARIABLES
//------------------------------------------------------------------------------
char* expandEnvironmentVariables(char* arg);
//------------------------------------------------------------------------------
// CHANGE DIRECTORY
//------------------------------------------------------------------------------
int changeDirectory(char* cmd);
//------------------------------------------------------------------------------
// EXIT
//------------------------------------------------------------------------------
int exitNicely(char* cmd);
//------------------------------------------------------------------------------
// PRINT WORKING DIRECTORY
//------------------------------------------------------------------------------
int printWorkingDirectory(char* cmd);
//------------------------------------------------------------------------------
// SET ENVIRONMENT VARIABLES
//------------------------------------------------------------------------------
// sets the environment variables passed
// in the command line
//
// Hints:
// - use 'block_contains()' to
//  get the index where the '=' is
// - 'get_environ_*()' can be useful here
void setEnvironmentVariables(char** eargv, int eargc);
//------------------------------------------------------------------------------
// RUN BACKGROUND
//------------------------------------------------------------------------------
void runBackground(struct cmd* cmd);
//------------------------------------------------------------------------------
// EXEC BACKGROUND
```

```
//-------------------------------------------------------------------------
int execBackground(struct cmd* cmd);
//-------------------------------------------------------------------------
// OPEN REDIR FD
//-------------------------------------------------------------------------
// opens the file in which the stdin/stdout or
// stderr flow will be redirected, and returns
// the file descriptor
//
// Find out what permissions it needs.
// Does it have to be closed after the execve(2) call?
//
// Hints:
// - if O_CREAT is used, add S_IWUSR and S_IRUSR
//  to make it a readable normal file
int openRedirFd(char* file);
//-------------------------------------------------------------------------
// RUN REDIR
//-------------------------------------------------------------------------
void runRedir(struct cmd* cmd);
//-------------------------------------------------------------------------
// RUN MULTIPLE PIPE
//-------------------------------------------------------------------------
void runMultiplePipe(struct cmd* cmd);
//-------------------------------------------------------------------------
// PARSING
//-------------------------------------------------------------------------
int parsing(struct execcmd* c, char* arg);
//-------------------------------------------------------------------------
// GET SIZE
//-------------------------------------------------------------------------
int getSize(char* cmd, char* commands[], int (*f)(char**, char*));
//-------------------------------------------------------------------------
// PRINT COMMAND
//-------------------------------------------------------------------------
int printCommand(char* commands[], size_t size);
//-------------------------------------------------------------------------
// FREE COMMAND
//-------------------------------------------------------------------------
int freeCommand(char* commands[], size_t size);
//-------------------------------------------------------------------------
// GET COMMANDS
//-------------------------------------------------------------------------
char** getCommands(char* cmd, size_t* size);
//-------------------------------------------------------------------------
// CREATE COMMANDS
```

```
//-----------------------------------------------------------------------------
struct cmd* createCommands(char* cmd);
//-----------------------------------------------------------------------------
#endif // FUNCTIONS_H
```

general.c

```
#include "general.h"
#include <errno.h>
#include <stdio.h>
#include <stdarg.h>
//-----------------------------------------------------------------------------
// PERR
//-----------------------------------------------------------------------------
void perr(const char *format, ...) {
    va_list args;
    va_start(args, format);
    char msgError[BUF_LEN];
    vsnprintf(msgError, BUF_LEN, format, args);
    va_end(args);
    perror(msgError);
}
//-----------------------------------------------------------------------------
```

general.h

```
#ifndef GENERAL_H
#define GENERAL_H
//-----------------------------------------------------------------------------
// INCLUDES
//-----------------------------------------------------------------------------
#include <unistd.h>
#include <dirent.h>
#define BUF_LEN 256
//-----------------------------------------------------------------------------
// PERR
//-----------------------------------------------------------------------------
// Pre: Recibo un formato y los argumentos
// Pos: Escribo el mensaje por pantalla y se muestra el mensaje de error
// correspondiente a la variable errno
```

```
void perr(const char *format, ...);
//--------------------------------------------------------------------------
#endif // GENERAL_H
```

parsing.c

```
#include "parsing.h"
#include "functions.h"
// parses an argument of the command stream input
static char* get_token(char* buf, int idx) {

    char* tok;
    int i;

    tok = (char*)calloc(ARGSIZE, sizeof(char));
    i = 0;

    while (buf[idx] != SPACE && buf[idx] != END_STRING) {
        tok[i] = buf[idx];
        i++; idx++;
    }

    return tok;
}

// parses and changes stdin/out/err if needed
static bool parse_redir_flow(struct execcmd* c, char* arg) {

    int inIdx, outIdx;

    if (parsing(c, arg)) return true;  // Challenge Parte 2: Flujo estándar
    // flow redirection for output
    if ((outIdx = block_contains(arg, '>')) >= 0) {
        switch (outIdx) {
            // stdout redir
            case 0: {
                strcpy(c->out_file, arg + 1);
                break;
            }
            // stderr redir
            case 1: {
                strcpy(c->err_file, &arg[outIdx + 1]);
                break;
```

```
            }
        }

        free(arg);
        c->type = REDIR;

        return true;
    }

    // flow redirection for input
    if ((inIdx = block_contains(arg, '<')) >= 0) {
        // stdin redir
        strcpy(c->in_file, arg + 1);

        c->type = REDIR;
        free(arg);

        return true;
    }

    return false;
}

// parses and sets a pair KEY=VALUE
// environment variable
static bool parse_environ_var(struct execcmd* c, char* arg) {

    // sets environment variables apart from the
    // ones defined in the global variable "environ"
    if (block_contains(arg, '=') > 0) {

        // checks if the KEY part of the pair
        // does not contain a '-' char which means
        // that it is not a environ var, but also
        // an argument of the program to be executed
        // (For example:
        //  ./prog -arg=value
        //  ./prog --arg=value
        // )
        if (block_contains(arg, '-') < 0) {
            c->eargv[c->eargc++] = arg;
            return true;
        }
    }

    return false;
```

```
}

// this function will be called for every token, and it should
// expand environment variables. In other words, if the token
// happens to start with '$', the correct substitution with the
// environment value should be performed. Otherwise the same
// token is returned.
//
// Hints:
// - check if the first byte of the argument
//  contains the '$'
// - expand it and copy the value
//   to 'arg'
static char* expand_environ_var(char* arg) {
    return expandEnvironmentVariables(arg);  // Your code here
}

// parses one single command having into account:
// - the arguments passed to the program
// - stdin/stdout/stderr flow changes
// - environment variables (expand and set)
static struct cmd* parse_exec(char* buf_cmd) {

    struct execcmd* c;
    char* tok;
    int idx = 0, argc = 0;

    c = (struct execcmd*)exec_cmd_create(buf_cmd);

    while (buf_cmd[idx] != END_STRING) {

        tok = get_token(buf_cmd, idx);
        idx = idx + strlen(tok);

        if (buf_cmd[idx] != END_STRING)
            idx++;

        tok = expand_environ_var(tok);

        if (parse_redir_flow(c, tok))
            continue;

        if (parse_environ_var(c, tok))
            continue;

        c->argv[argc++] = tok;
```

```
    }

    c->argv[argc] = (char*)NULL;
    c->argc = argc;

    return (struct cmd*)c;
}

// parses a command knowing that it contains
// the '&' char
static struct cmd* parse_back(char* buf_cmd) {

    int i = 0;
    struct cmd* e;

    while (buf_cmd[i] != '&')
        i++;

    buf_cmd[i] = END_STRING;

    e = parse_exec(buf_cmd);

    return back_cmd_create(e);
}

// parses a command and checks if it contains
// the '&' (background process) character
struct cmd* parse_cmd(char* buf_cmd) {

    if (strlen(buf_cmd) == 0)
        return NULL;

    int idx;

    // checks if the background symbol is after
    // a redir symbol, in which case
    // it does not have to run in in the 'back'
    if ((idx = block_contains(buf_cmd, '&')) >= 0 && buf_cmd[idx - 1] != '>' &&
            buf_cmd[idx + 1] != '>')
        return parse_back(buf_cmd);

    return parse_exec(buf_cmd);
}

// parses the command line
// looking for the pipe character '|'
```

```
struct cmd* parse_line(char* buf) {
    return createCommands(buf);
}
```

parsing.h

```
#ifndef PARSING_H
#define PARSING_H

#include "defs.h"
#include "types.h"
#include "createcmd.h"
#include "utils.h"
struct cmd* parse_cmd(char* buf_cmd);
struct cmd* parse_line(char* b);

#endif // PARSING_H
```

printstatus.c

```
#include "printstatus.h"

// prints information of process' status
void print_status_info(struct cmd* cmd) {

    if (strlen(cmd->scmd) == 0
        || cmd->type == PIPE)
        return;

    if (WIFEXITED(status)) {

        fprintf(stdout, "%s Program: [%s] exited, status: %d %s\n",
            COLOR_BLUE, cmd->scmd, WEXITSTATUS(status), COLOR_RESET);
        status = WEXITSTATUS(status);

    } else if (WIFSIGNALED(status)) {

        fprintf(stdout, "%s Program: [%s] killed, status: %d %s\n",
            COLOR_BLUE, cmd->scmd, -WTERMSIG(status), COLOR_RESET);
```

```
        status = -WTERMSIG(status);

    } else if (WTERMSIG(status)) {

        fprintf(stdout, "%s Program: [%s] stopped, status: %d %s\n",
            COLOR_BLUE, cmd->scmd, -WSTOPSIG(status), COLOR_RESET);
        status = -WSTOPSIG(status);
    }
}

// prints info when a background process is spawned
void print_back_info(struct cmd* back) {

    fprintf(stdout, "%s  [PID=%d] %s\n",
        COLOR_BLUE, back->pid, COLOR_RESET);
}
```

printstatus.h

```
#ifndef PRINTSTATUS_H
#define PRINTSTATUS_H

#include "defs.h"
#include "types.h"

extern int status;

void print_status_info(struct cmd* cmd);

void print_back_info(struct cmd* back);

#endif // PRINTSTATUS_H
```

readline.c

```
#include "defs.h"
#include "readline.h"
```

```
static char buffer[BUFLEN];

// read a line from the standar input
// and prints the prompt
char* read_line(const char* promt) {

    int i = 0,
        c = 0;

    fprintf(stdout, "%s %s %s\n", COLOR_RED, promt, COLOR_RESET);
    fprintf(stdout, "%s", "$ ");

    memset(buffer, 0, BUFLEN);

    c = getchar();

    while (c != END_LINE && c != EOF) {
        buffer[i++] = c;
        c = getchar();
    }

    // if the user press ctrl+D
    // just exit normally
    if (c == EOF)
        return NULL;

    buffer[i] = END_STRING;

    return buffer;
}
```

readline.h

```
#ifndef READLINE_H
#define READLINE_H

char* read_line(const char* promt);

#endif //READLINE_H
```

runcmd.c

```
#include "runcmd.h"
#include "functions.h"
int status = 0;
struct cmd* parsed_pipe;

// runs the command in 'cmd'
int run_cmd(char* cmd) {

    pid_t p;
    struct cmd *parsed;

    // if the "enter" key is pressed
    // just print the promt again
    if (cmd[0] == END_STRING)
        return 0;

    // cd built-in call
    if (cd(cmd))
        return 0;

    // exit built-in call
    if (exit_shell(cmd))
        return EXIT_SHELL;

    // pwd buil-in call
    if (pwd(cmd))
        return 0;

    // parses the command line
    parsed = parse_line(cmd);

    // forks and run the command
    if ((p = fork()) == 0) {

        // keep a reference
        // to the parsed pipe cmd
        // so it can be freed later
        if (parsed->type == PIPE)
            parsed_pipe = parsed;

        exec_cmd(parsed);
    }
```

```
    // store the pid of the process
    parsed->pid = p;

    // background process special treatment
    // Hint:
    // - check if the process is
    //  going to be run in the 'back'
    // - print info about it with
    //  'print_back_info()'
    //
    // Your code here
    // waits for the process to finish
    if (!execBackground(parsed)) waitpid(p, &status, 0);

    print_status_info(parsed);

    free_command(parsed);

    return 0;
}
```

runcmd.h

```
#ifndef RUNCMD_H
#define RUNCMD_H

#include "defs.h"
#include "parsing.h"
#include "exec.h"
#include "printstatus.h"
#include "freecmd.h"
#include "builtin.h"

int run_cmd(char* cmd);

#endif // RUNCMD_H
```

sh.c

```
#include "defs.h"
#include "types.h"
#include "readline.h"
#include "runcmd.h"

char promt[PRMTLEN] = {0};

// runs a shell command
static void run_shell() {

    char* cmd;

    while ((cmd = read_line(promt)) != NULL)
        if (run_cmd(cmd) == EXIT_SHELL)
            return;
}

// initialize the shell
// with the "HOME" directory
static void init_shell() {

    char buf[BUFLEN] = {0};
    char* home = getenv("HOME");

    if (chdir(home) < 0) {
        snprintf(buf, sizeof buf, "cannot cd to %s ", home);
        perror(buf);
    } else {
        snprintf(promt, sizeof promt, "(%s)", home);
    }
}

int main(void) {

    init_shell();

    run_shell();

    return 0;
}
```

types.h

```
#ifndef TYPES_H
#define TYPES_H

#include "defs.h"

/* Commands definition types */

/*
 cmd: Generic interface
    that represents a single command.
    All the other *cmd structs can be
    casted to it, and they don´t lose
    information (for example the 'type' field).

    - type: {EXEC, REDIR, BACK, PIPE}
    - pid: the process id
    - scmd: a string representing the command before being parsed
*/
struct cmd {
    int type;
    pid_t pid;
    char scmd[BUFLEN];
};

/*
  execcmd: It contains all the relevant
    information to execute a command.

    - type: could be EXEC or REDIR
    - argc: arguments quantity after parsed
    - eargc: environ vars quantity after parsed
    - argv: array of strings representig the arguments
    - eargv: array of strings of the form: "KEY=VALUE"
        representing the environ vars
    - *_file: string that contains the name of the file
        to be redirected to

    IMPORTANT: an execcmd struct can have EXEC or REDIR type
        depending on if the command to be executed
        has at least one redirection symbol (<, >, >>, >&)
*/
struct execcmd {
    int type;
```

```
    pid_t pid;
    char scmd[BUFLEN];
    int argc;
    int eargc;
    char* argv[MAXARGS];
    char* eargv[MAXARGS];
    char out_file[FNAMESIZE];
    char in_file[FNAMESIZE];
    char err_file[FNAMESIZE];
};

/*
  pipecmd: It contains the same information as 'cmd'
    plus two fields representing the left and right part
    of a command of the form: "command1 arg1 arg2 | command2 arg3"
    As they are of type 'struct cmd',
    it means that they can be either an EXEC or a REDIR command.
*/
struct pipecmd {
    int type;
    pid_t pid;
    char scmd[BUFLEN];
    struct cmd** cmdVec;
    size_t size;
};

/*
  backcmd: It contains the same information as 'cmd'
    plus one more field containing the command to be executed.
    Take a look to the parsing.c file to understand it better.
    Again, this extra field, can have type either EXEC or REDIR
    depending on if the process to be executed in the background
    contains redirection symbols.
*/
struct backcmd {
    int type;
    pid_t pid;
    char scmd[BUFLEN];
    struct cmd* c;
};

#endif // TYPES_H
```

utils.c

```
#include "utils.h"

// splits a string line in two
// acording to the splitter character
char* split_line(char* buf, char splitter) {

    int i = 0;

    while (buf[i] != splitter &&
           buf[i] != END_STRING)
        i++;

    buf[i++] = END_STRING;

    while (buf[i] == SPACE)
        i++;

    return &buf[i];
}

// looks in a block for the 'c' character
// and returns the index in which it is, or -1
// in other case
int block_contains(char* buf, char c) {

    for (int i = 0; i < strlen(buf); i++)
        if (buf[i] == c)
            return i;

    return -1;
}
```

utils.h

```
#ifndef UTILS_H
#define UTILS_H

#include "defs.h"

char* split_line(char* buf, char splitter);
```

```
int block_contains(char* buf, char c);

#endif // UTILS_H
```