By Hacman Viorica Gabriela
MINISHELL

Bootstrap – fase zero → tutto ciò che serve per accendere il programma:
Il main prepara la struttura principale t_shell, inizializza lo stato (azzerando l'exit_status, impostando i puntatori a null, copiando il nome del programma, etc). Configura i segnali in modalità prompt.
SIGINT (Ctrl-C) viene gestito con un handler che non uccide il processo, ma "ripulisce" la riga corrente e ripresenta il prompt. SIGQUIT viene **ignorato** mentre si è al prompt.

Poi chiama il loop interattivo: start_colored_prompt(&shell). Alla fine libera le risorse globali con destroy_shell.

Loop interattivo (prompt_color.c)

start_colored_prompt esegue un setup "interattivo" dei segnali (coerente con quanto impostato in main) e entra in un **while(1)** che richiama run_prompt_once(shell).

Il ciclo si interrompe quando run_prompt_once segnala che bisogna uscire (ad esempio su EOF/Ctrl-D o su comando di uscita). In quel caso, start_colored_prompt ritorna l'exit_status corrente, che diventa il codice di uscita del programma.

Una iterazione del prompt (prompt_color.c / prompt_helper.c)

Per ogni giro, si costruisce la **stringa del prompt colorato** (tipicamente includendo info come directory corrente o stato dell'ultimo comando) con una funzione del tipo create_colored_prompt(...).

Si invoca readline(prompt).

Qui ci sono tre casi tipici:

1. **EOF/Ctrl-D**: readline restituisce NULL → si esce dal loop (e quindi dalla shell) in modo pulito.
2. **Linea vuota o solo spazi**: in genere si scarta e si ripete il ciclo.
3. **Linea valida**: si aggiunge alla history (se non vuota) e la si passa alla fase di gestione input.

Gestione dell'input (input_handler.c)

Prima di "capire" la riga, il codice tenta di **completarla** se è sintatticamente "aperta": esempio classico, **apici non chiusi**. Questo è il ruolo di una funzione come get_complete_input(...), che continua a leggere finché non si chiudono le quote.

Ottenuta la riga completa, parte la **tokenizzazione** con parse_line_to_tokens(complete_input, shell). Qui si scorporano parole, operatori (|, <, >, >>, <<), si gestiscono quoting e – tipicamente – le espansioni (variabili d'ambiente, ecc.) secondo la semantica che è stata implementata.

Subito dopo c'è un **controllo di sintassi** mirato sulle **pipe** (check_syntax_pipes(tokens)): cattura errori comuni (pipe a inizio/fine linea, doppie pipe senza comando in mezzo, ecc.). Se fallisce, lo exit_status viene impostato a 2 (errore di sintassi) e si interrompe l'elaborazione di quella riga.

Se la sintassi è ok, si entra nella fase di **processamento dei token** (process_tokens(tokens, shell)), dove tipicamente:

1. si costruisce la/e **pipeline** di comandi strutturandoli in una lista/array di "comandi" con i loro argomenti;
2. si distinguono **built-in** da comandi esterni;
3. si preparano **redirezioni** e **here-doc**;
4. si esegue la pipeline: per pipe multiple si forka, si collega l'I/O con le pipe, si applicano redirezioni per ogni processo, e si attende/finalizza aggiornando exit_status.

Al termine della riga, si fa la **pulizia per-comando** (cleanup_per_command(shell)), liberando token, strutture dei comandi e ogni risorsa temporanea.

Gestione dei segnali al prompt (prompt_helper.c)

L'handler di SIGINT (Ctrl-C) è "amichevole con readline": stampa una newline per andare a capo, avvisa readline che inizia una nuova riga (rl_on_new_line), **sostituisce** la riga corrente con vuoto (rl_replace_line("", 0)) e **ridisegna** il prompt (rl_redisplay). Risultato: non chiude la shell e non manda in background, ma interrompe l'editing della riga e ti riporta subito al prompt.

SIGQUIT in modalità prompt viene ignorato, così non butta fuori messaggi tipo "Quit (core dumped)" mentre stai scrivendo.

Com'è organizzato il flusso logico

Inizializzazione → setup segnali → loop.

Ogni giro del loop: **costruisci prompt → leggi riga → (se necessario) completa riga → tokenizza → valida sintassi (pipe) → costruisci pipeline/gestisci redirezioni → esegui → aggiorna exit_status → pulisci → ripeti.**

Uscita pulita su EOF/Ctrl-D o su condizione di uscita segnalata da run_prompt_once.

Schema mentale del flusso

Bootstrap → segnali → loop
Ogni iterazione:

1. costruisci **prompt**
2. **readline**
3. (se serve) **completa** la riga
4. **tokenizza**
5. **valida** la sintassi (pipe)
6. **costruisci** pipeline / **redirezioni**
7. **esegui** e aggiorna **exit_status**
8. **pulisci**
9. ripeti

LEXER

Ingresso e contesto

La funzione d'ingresso è parse_line_to_tokens(const char *str, t_shell *shell). Prepara un **t_token_context** con:

input (la riga intera), un indice intero passato **per indirizzo** (int *i), così le funzioni avanzano direttamente il cursore, il puntatore alla **lista dei token** (t_token **tokens), shell (per eventuali usi interni).

Inizializza anche un flag had_whitespace, che indica se **prima del prossimo token** è stato visto spazio bianco. Questo dettaglio è importante perché il lexer può dover **unire segmenti adiacenti senza spazi** (es. foo"bar" dev'essere un'unica parola foobar).

Ciclo principale di scansione

Il lavoro vero lo fa run_tokenization_loop(...). Ad ogni iterazione: chiama skip_whitespace_and_check(...) che **salta spazi/tab** aggiornando l'indice e, se ha saltato qualcosa, imposta had_whitespace = 1; se non siamo a fine stringa, memorizza had_whitespace nel contesto e chiama process_next_token(...); subito dopo, rimette had_whitespace = 0 (così "bianco" vale solo per quel token appena iniziato). Il loop termina quando non ci sono più caratteri da

leggere. In caso di errore durante l'estrazione di un token, ritorna fallimento e parse_line_to_tokens libera la lista parziale.

Come riconosce i token

La funzione **chiave** è process_next_token(t_token_context *context). Guarda il carattere corrente e decide cosa fare:

Apici/virgolette (' o "): chiama handle_quoted_token(context). Questa routine legge una **sottostringa quotata** completa. Per determinare la fine: usa find_closing_quote(...), e in caso di " passa per process_escaped_content(...) che interpreta le **sequenze di escape** (vedi sotto). in caso di ' prende tutto alla lettera (nessuna escape). Restituisce un **segmento** che poi verrà: o creato come nuovo token, oppure **concatenato** al valore dell'ultimo token *se non c'è stato spazio prima* (grazie al flag had_whitespace). Esempio: in echo foo"bar", foo + "bar" diventano un **unico** token foobar.

- **Redirezioni** (< o >): chiama handle_redirection_token(...). Qui si invoca get_redir(...) che controlla il carattere successivo per distinguere:

<< → TK_HEREDOC, >> → TK_APPEND, > → TK_OUT, < → TK_IN.

Avanza l'indice di 1 o 2 caratteri a seconda del match e **crea il token** con il tipo giusto e la stringa corrispondente ("<<", ">>", ">", "<").

- **Pipe** (|): crea direttamente un token TK_PIPE con valore "|" e avanza l'indice di 1.
- **Altrimenti → parola**: chiama handle_word_token(context).

Questa funzione legge una **word** fino al prossimo **separatore**. I separatori (definiti da is_separator(...)) includono spazi/tab e metacaratteri (|, <, >).

Anche qui si rispetta la regola di **unione** con il token precedente se had_whitespace == 0 (es.: abc$VAR o abc"def" diventano un token unico).

In tutte le creazioni di token si usa create_token(value, type) che:

alloca un t_token, duplica la stringa value, imposta il type e next = NULL. L'inserimento in lista è con add_token_to_list(&tokens, new_token).

Spazi bianchi e separatori

skip_whitespace(...) (usato da skip_whitespace_and_check) **conta e salta** spazi/tab partendo dalla posizione corrente; l'indice principale *i viene poi aggiornato.

skip_whitespace_and_check(...) inoltre: **setta had_whitespace** se ha saltato qualcosa, ritorna 0/1 per dire se c'è ancora input da leggere. only_spaces(...) è un comodo check per vedere se una stringa è fatta *solo* di spazi/tab (utile per scartare linee vuote ai fini della history).

Virgolette ed escape (file escape.c)

Qui è gestita tutta la parte "delicata": has_unclosed_quotes(str) serve **a monte** (viene usata anche per l'input multilinea nel completamento riga) e controlla se ci sono apici/virgolette **aperte ma non chiuse**; in quel caso l'acquisizione dell'input continua su più righe (readline("> ")) finché non chiudi. find_closing_quote(str, start, quote_char) trova la posizione della chiusura; in caso di mancanza restituisce -1. process_escaped_content(...) interpreta le sequenze \n, \t, \r, \\, \", \' **solo per contenuti in doppi apici** ("). Con apici singoli (') non si tocca

nulla: il contenuto è **letterale**. extract_quoted_substring(str, index, quote_char) mette insieme il tutto: salta l'apertura, cerca la chiusura, se non c'è, segnala "fino a fine stringa" (il chiamante potrà gestire l'errore o far proseguire il completamento riga), estrae il contenuto correttamente (con o senza escape, a seconda del tipo di quote), aggiorna *index alla posizione subito dopo la chiusura. Questa separazione ti garantisce che casi come:

- echo "a\nb"
- echo 'a\nb'
  vengano distinti: nel primo caso \n diventa newline reale, nel secondo resta \+n.

Redirezioni (file token_redirection.c)

get_redir_type_and_length(first, second, &length) decide il **tipo** e quanti caratteri consumare:

<< → TK_HEREDOC (length 2),

>> → TK_APPEND (length 2),

< → TK_IN (length 1),

> → TK_OUT (length 1).

get_redir(...) incapsula questa logica, **avanza l'indice** e restituisce anche la **stringa canonica** del token ("<<", ">>", ">", "<").

handle_redirection_token(...) crea il token e lo mette in lista.

Nota: qui si crea **solo l'operatore**. L'**argomento** della redirezione (file/limitatore heredoc) verrà letto come **word** dal passo successivo del lexer.

Utility di supporto (file token_lexer_utils.c)

Funzioni per **input multilinea**: ft_trim_spaces(...) (se presente) e soprattutto get_complete_input(initial_input) che:

duplica l'input iniziale, finché has_unclosed_quotes(...) è vero, chiama readline("> "), ogni riga aggiuntiva viene concatenata con append_line_to_input(...) che inserisce anche un newline tra i pezzi. Risultato: quando arriva al lexer, la stringa è **completa** (quote chiuse).

Perché il flag had_whitespace è importante

Il flag viene passato nel contesto prima di "iniziare" un token e **vale solo per quel token**. Serve a **decidere se aprire un nuovo token o concatenare** al precedente quando due segmenti lessicali sono adiacenti **senza spazi**.
Esempi pratici:

- echo foo"bar" → un solo token foobar (word + quoted segment, senza spazio).
- echo "foo""bar" → un solo token foobar (due quoted adiacenti, senza spazio).
- echo foo "bar" → due token separati foo e bar (perché c'è spazio in mezzo).

Separazione chiara dei ruoli

- **Lexer**: riconosce strutture sintattiche elementari (pipe, operatori di redirezione, parole, segmenti quotati) e produce una **lista tipizzata** di token, rispettando le regole su spazi/quote/escape.

- **Parsing/Esecuzione**: più avanti si costruiscono pipeline, si associano **argomenti ai comandi**, si attribuiscono **redirezioni ai comandi giusti**, si gestisce l'heredoc, ecc.

Edge case gestiti:

- **Quote non chiuse**: l'acquisizione continua su più righe finché chiudi (niente token "mozzati").
- **Escape in doppi apici**: tradotte in caratteri reali; nei singoli apici no.
- **Operatori doppi** (<<, >>) riconosciuti correttamente e distinti dai singoli (<, >).
- **Concatenazione senza spazi**: preservata, così non perdi semantica negli argomenti.

PARSER

Il parser parte dalla **lista tipizzata di token** prodotta dal lexer (WORD, PIPE, IN, OUT, APPEND, HEREDOC, ...) e ha tre macro-responsabilità:

1. **Rendere i WORD semanticamente corretti** per la shell: unescape dove serve, **espandere le variabili** e preservare il significato delle quote.
2. **Convalidare la sintassi delle redirezioni** prima di costruire i comandi, così non si propaga stato incoerente all'esecutore.
3. **Segmentare in pipeline e costruire comandi**: tagliare per |, montare argv[], raccogliere le redirezioni e filtrare gli argomenti "rumore" post-espansione.

Il filo del discorso (control flow)

1. **Token WORD → Parola logica (con espansioni)**
   • handle_word_token(context) chiama
   → extract_and_expand_word(input, i, shell).
   • Se la parola deriva da sezioni quotate:
   → process_quoted_content(input, i, shell) (in token_parser.c) estrae il segmento; se è **"..."**, applica expand_variables(...); se è **'...'**, lascia letterale.
   • I frammenti adiacenti **senza spazio** si **congiungono**: append_to_previous_word_token(context, expanded) e append_quoted_to_last_word_token(...) mantengono l'invariante "**un solo token WORD per argomento logico**".
2. **Espansione variabili (dentro WORD e dentro "...")**
   • expand_variables(...) scorre la stringa e quando vede $ invoca process_dollar_sign(...).
   • Casi coperti: $NAME, ${NAME}, $? (usa shell->exit_status), e **$0** con expand_program_name(...).
   • L'estrazione del nome variabile e la lookup avvengono in env_expansion.c (con find_env_node(...) su shell->env e fallback a getenv(...)). Se mancante → stringa vuota.

se in extract_raw_word(...) trovi * **non escape-ata**, la trasformi in "*" (vedi controllo ft_strchr(word, '*') && !ft_strstr(word, "*")) per **disinnescare il globbing**; in seguito l'unescape rimetterà * letterale. È una scelta semplice e pulita per rispettare i vincoli del progetto (nessun globbing).

3. **Unescape mirato**
   • La rimozione dei backslash fuori dalle quote è gestita da utility in parser_utils.c (es. una funzione tipo unescape_unquoted(...) e un'helper stile unescape_copy(...) che copia saltando \x).
   • Invarianti:
   – Dentro '...' non si tocca nulla.
   – Dentro "..." l'espansione **è attiva**, ma l'unescape segue le regole consentite.
   – Fuori dalle quote, \ rimuove il significato speciale del char successivo.

4. **Validazione redirezioni (prima di costruire)**
   • validate_input_redirections(tokens) chiama:
   → validate_in_redirections(tokens) per < e
   → validate_heredoc_redirections(tokens) per <<.
   • Simmetricamente (stesso file) convalida > e >>.
   ogni operatore dev'essere **immediatamente** seguito da un TK_WORD valido (filename o delimitatore heredoc). Sequenze tipo cmd > | o cmd << senza delimitatore vanno in errore e il parser ritorna con exit_status = 2.
   ARRIVA al "builder" solo con token stream coerente.

5. **Segmentazione per pipeline**
   • populate_command_args(cmd, token_start) usa
   → count_args_until_pipe(...) per sapere la dimensione argv,
   → alloc_args_array(...) e poi
   → fill_args_until_pipe(...) per copiare i valori fino al TK_PIPE/fine.
   In parallelo, il builder scorre gli stessi token e **non mette** nei argv gli operatori di redirezione, ma **cattura** i loro **operand** come metadati I/O del t_cmd.

6. **Costruzione dei comandi e della pipeline**
   • process_single_command(&current_token, shell, &head, &tail) (in pipe_command_builder.c) è il cuore: parte da current_token, consuma un "troncone" fino a | o fine,
   – alloca t_cmd,
   – popola argv (saltando i metacaratteri) e
   – registra redirezioni (IN, OUT, APPEND, HEREDOC) con i rispettivi target.
   • extract_command_tokens(...) orchestra questo processo e restituisce la **lista collegata di t_cmd**, una per ciascun segmento tra pipe. L'ordine si mantiene naturale (testa → coda della pipeline).

7. **Filtraggio argomenti post-espansione**
   • build_clean_args_array(tokens) (nella tua organizzazione: count_non_redirection_words(...), copy_args_to_array(...)) costruisce un argv[] **coerente** con la semantica delle quote:
   – **Argomenti vuoti quotati** (es. "") vanno **preservati**.
   – **Vuoti non quotati** derivati da espansioni (es. $UNSET) possono essere **rimossi**.
   • Invarianti: argv termina **sempre** con NULL, e argv[0] è definito solo se il comando ha almeno un argomento "valido".

Invarianti che il parser mantiene

Coesione lexicale → semantica: i frammenti senza spazio si uniscono sempre nello **stesso** TK_WORD (append_to_previous_word_token, append_quoted_to_last_word_token). **Quote forti vs deboli**: '...' blocca espansioni; "..." le abilita. **No globbing**: * non escape-ato viene neutralizzato (trasformato in * e poi unescape → * letterale). **Redirezioni sane**: ogni operatore ha **un solo** target "parola" immediatamente successivo (validato in redirect_validation.c). **Pipeline ben formata**: mai | in testa/coda o doppie pipe senza comando in mezzo (controlli in fase di slicing/conta segmenti). **argv pulito**: nessun metacarattere, solo gli argomenti "veri", con i vuoti gestiti secondo regole shell.

Focus sulle espansioni: come "legge" i dollari

- **Scansione**: expand_variables(str, shell) percorre la stringa; a ogni $ delega a process_dollar_sign(str, i, shell).
- **$?**: sostituito con shell->exit_status.
- **$0**: expand_program_name(...) mette il nome del programma (salvato nel bootstrap).
- **$VAR / ${VAR}**: estrazione del nome (lettere, cifre, underscore), lookup in shell->env (find_env_node) con fallback a getenv. Se non trovata → "".

Costruzione di un comando: che cosa succede "dentro"

Dentro process_single_command(...) si può leggere così:

1. Si identifica l'intervallo [start, before_next_pipe) sulla lista di token.
2. **Prima passata**: validazione locale delle redirezioni su quel range (se non già globalizzata).
3. **Seconda passata**:
   – per ogni TK_WORD → push in argv;
   – per ogni TK_IN/TK_OUT/TK_APPEND/TK_HEREDOC → prendi il **token successivo** (deve essere WORD) e registra la redirezione nel t_cmd.
4. populate_command_args(cmd, start) crea argv dimensionato giusto (count_args_until_pipe).
5. Se TK_PIPE era presente, collega il t_cmd appena creato con il prossimo nella pipeline.

Risultato: una lista t_cmd **lineare** e già completa (argv + IO metadata), pronta per l'exec/fork stage.

EXECUTOR

Dopo il parser il programma riceve una **pipeline di comandi** (t_cmd): ciascun nodo ha argv, eventuali redirezioni (IN, OUT, APPEND, HEREDOC) e un link al prossimo. L'executor deve:

1. **Creare il grafo dei file descriptor** (pipe e redirezioni),
2. **Applicare l'ordine corretto** tra redirezioni (l'ultima vince),
3. **Gestire gli heredoc** rispettando il quoting del delimitatore,
4. **Decidere dove eseguire i built-in** (nel padre o nel figlio),
5. **Ricercare il binario** su $PATH e lanciare execve,
6. **Propagare i segnali** correttamente (prompt vs esecuzione),
7. **Pulire le risorse** e **ritornare un exit status** coerente.

1) Preparazione dei dati e dell'ambiente (prepare/)

exec_prepare.c: punto d'ingresso dell'esecuzione; coordina l'intero ciclo. args_convert.c: qualsiasi lista interna diventa char **argv con terminatore NULL, perché execve lo richiede.

pipe_enviroment.c: serializza la tua lista t_env in char **envp ("KEY=VALUE"). Importante: in pipeline i figli erediteranno questa fotografia dell'ambiente al momento del fork. **Perché serve**: execve non sa nulla delle strutture; vuole array C classici. Questo "translation layer" separa l'API esterna (execve) dal model (liste).

2) Segnali: tre profili (signal/)

- signal_setup.c definisce **modalità**:
  - **Prompt mode**: SIGINT non uccide, pulisce la riga (readline).
  - **Executor mode**: SIGINT/SIGQUIT non sono ignorati: i figli ricevono i segnali; il padre **non** stampa artefatti del prompt.
  - **Heredoc mode**: SIGINT interrompe la lettura del documento.
- signals.c, signal_utils.c: funzioni di supporto e decodifica waitpid — WIFEXITED / WIFSIGNALED. Se un figlio muore per segnale, si adotta la convenzione **bash-like**: exit_status = 128 + sig.

Perché serve: l'interattività richiede comportamenti diversi. Durante l'esecuzione non bisogna "sporcare" l'output; al prompt invece vuoi che il programma resti in vita e ridisegnare.

3) Redirezioni e heredoc (redirection/)

- redirect_handler.c: "entry point" che applica le redirezioni di un comando. Chiama:
  - redirect_input.c (e redir_in_setup.c) per < e <<,
  - redirect_output.c per > e >>,
  - redirect_utils.c per open sicuri, dup2, check permessi.

se sono presenti **più redirezioni dello stesso tipo**, prevale **l'ultima** (comportamento delle shell).

heredoc.c: legge fino al **delimitatore**. Se il delimitatore è **quotato**, il corpo **non** viene espanso; se non è quotato, puoi espandere variabili (la tua implementazione conserva l'informazione sul quoting del delimitatore per decidere).

il grafo dei fd del processo figlio nasce qui. Prima si "cablano" redirezioni e heredoc, poi eventualmente si collegano le pipe.

4) Built-in: dove e quando (builtin/ + process/)

- handle_built_in.c: riconosce se argv[0] è un builtin.
- exec_builtins.c: **decide il contesto** di esecuzione:
    - Se **singolo comando** e builtin "di stato" (cd, export, unset, exit) → **nel padre** (senza fork), perché devono modificare directory o ambiente del processo corrente.
    - Se in **pipeline** o builtin "stateless" (echo, pwd, env) → **nel figlio**.
- Moduli specifici:
    - built_in_cd/*: cambia directory, aggiorna PWD/OLDPWD (via env_setters.c/env_set_utils.c).
    - built_in_export/*: parsing di VAR/VAR=val, inserimento/ aggiornamento in t_env, stampa ordinata senza argomenti.
    - builtin_unset.c, built_in_env/*, builtin_echo.c, builtin_pwd.c.
    - built_in_exit/*: validazione argomento numerico e semantica errori.

Perché serve: distinguere **padre** vs **figlio** evita "false modifiche": se export corre in un figlio di pipeline, l'ambiente del padre **non** cambia.

5) Ricerca del binario (command_path/)

- env_path_split.c, path_split_utils.c: estraggono $PATH, lo splittano, normalizzano le directory (rimuovere vuoti, aggiungere /).
- path_search.c:
    - Se argv[0] contiene / → prova execve diretto (path assoluto o relativo).
    - Altrimenti itera su $PATH con access(X_OK) finché trova un eseguibile; se non trovato → "command not found" e status 127.

Perché serve: la risoluzione del comando dev'essere veloce e deterministica, e dare messaggi coerenti (permessi vs inesistente).

6) Pipeline: "grafo" dei processi (pipeline/ + process/)

- pipe_setup.c, pipe_utils.c: alloca tutte le pipe **prima** del ciclo dei fork (per N comandi servono N-1 pipe).
- fork_children.c: per ogni t_cmd:
    - **fork**; nel figlio si passa a pipe_child_setup.c e pipe_child_IO.c.
- Nel **figlio**:
    - pipe_child_setup.c collega stdin/stdout alle **estremità corrette** della pipe (primo, intermedio, ultimo).
    - Poi redirect_handler.c applica le **redirezioni del comando**: l'ordine tipico è pipe-IO → redirezioni (ma la tua implementazione garantisce che **l'ultima redirezione** ancora sovrascriva).
    - Infine: builtin nel figlio **oppure** path_search.c + execve.
- Nel **padre**:
    - pipe_handler.c/pipe_execution.c chiudono tempestivamente i fd non più necessari (evita deadlock) e chiamano process_handler.c per i waitpid multipli. L'**ultimo** comando della pipeline determina $? (comportamento bash).

Perché serve: la pipeline è un **grafo lineare** di processi con accoppiamenti fd → fd; l'ordine delle chiusure è cruciale per non rimanere appesi.

cat < in.txt | grep foo > out.txt

Supponiamo che il parser ti consegni tre t_cmd:

- **Cmd0**: argv = ["cat"], redir: IN -> "in.txt"
- **Cmd1**: argv = ["grep", "foo"], nessuna redir
- **Cmd2**: argv = ["tee"]? No, nel tuo esempio è **solo** redir OUT: argv = ["grep","foo"] è nel cmd1; **Cmd2** è in realtà il **secondo** e **ultimo** comando: argv = ["grep","foo"], redir: OUT -> "out.txt"

pipeline di **due** comandi:

1. cat < in.txt
2. grep foo > out.txt

Flusso:

1. **Preparazione**
    - exec_prepare.c costruisce envp (pipe_enviroment.c) e prepara l'array di pipe: per 2 comandi → **1 pipe** (pipe_setup.c).
2. **Creazione pipe**
    - pipe_setup.c chiama pipe(fd[2]) → supponiamo fd = {p[0], p[1]}.
3. **Fork del primo comando** (fork_children.c)
    - **Figlio #1** (cat):
        - pipe_child_setup.c: è il **primo** della pipeline → collega STDOUT_FILENO a **p[1]** con dup2(p[1], 1); chiude p[0].
        - redirect_handler.c: applica IN -> "in.txt": open("in.txt", O_RDONLY) → dup2(fd_in, 0), chiude fd_in.
        - Chiude i fd di pipe rimasti (ordinatamente).
        - Non è builtin "di stato" → path_search.c trova /bin/cat → execve.
    - **Padre**: chiude **p[1]** (lato scrittura) subito dopo il fork del primo figlio (importante per non trattenere la pipe aperta).
4. **Fork del secondo comando**
    - **Figlio #2** (grep foo > out.txt):
        - pipe_child_setup.c: è l'**ultimo** della pipeline → collega STDIN_FILENO a **p[0]** con dup2(p[0], 0); chiude p[1].
        - redirect_handler.c: applica OUT -> "out.txt": open("out.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) → dup2(fd_out, 1), chiude fd_out.
        - Chiude i fd residui.
        - Non è builtin → path_search.c trova /bin/grep → execve.
    - **Padre**: chiude **p[0]** (lato lettura) dopo il fork del secondo figlio.
5. **Attesa e status**
    - process_handler.c: waitpid su entrambi; lo **status della pipeline** è quello dell'**ultimo** comando (grep). Se grep trova righe → 0, se non trova → 1. signal_utils.c traduce eventuali terminazioni per segnale.

6. **Risultato**
    - cat legge da in.txt, scrive su pipe; grep legge dalla pipe, filtra “foo”, scrive in out.txt.

Scelte logiche importanti (e perché sono corrette)

- **Ultima redirezione vince**: allinei il comportamento a bash; evita ambiguità quando l’utente scrive cose come cmd > a > b.
- **Padre vs figlio per i built-in**: conservi lo stato corretto dell’ambiente e della working directory del processo interattivo.
- **Chiusura anticipata dei fd nel padre**: è essenziale. Se il padre conserva un’estremità aperta, i figli possono restare bloccati in lettura (EOF non arriva mai).
- **Status della pipeline** = status dell’ultimo: compatibilità attesa dagli script (es. set -o pipefail non è richiesto in minishell).
- **Gestione segnali differenziata**: rende l’interfaccia utente pulita e prevedibile.

Error handling: cosa succede davvero

- redirect_utils.c centralizza gli errori di open()/dup2() con messaggi uniformi e exit_status coerente (tipicamente 1; 126 per permessi negati all’esecuzione; 127 per “command not found”).
- process_handler.c decodifica waitpid:
    - WIFEXITED → usa WEXITSTATUS,
    - WIFSIGNALED → 128 + WTERMSIG.
- path_search.c decide tra “not found” vs “permission denied” (access(F_OK) vs access(X_OK))