Minishell Parsing Guide

A complete reference for the parsing pipeline — mmubina @ 42

Table of Contents

Part I — Concepts You Need to Know

1. The Big Picture: Parsing Pipeline

When a user types a command in your shell, it goes through a series of transformations before the shell can execute it. Each stage takes one form of data and converts it to the next:

User types: `echo $HOME | cat -e > out.txt`

↓ *Stage 1: Tokenizer (Lexer)*

`[WORD:"echo"] → [WORD:"$HOME"] → [PIPE:"|"] → [WORD:"cat"] → [WORD:"-e"] → [REDIR_OUT:">"] → [WORD:"out.txt"]`

↓ *Stage 2: Expander*

`[WORD:"echo"] → [WORD:"/home/mmubina"] → [PIPE:"|"] → [WORD:"cat"] → [WORD:"-e"] → [REDIR_OUT:">"] → [WORD:"out.txt"]`

↓ *Stage 3: Syntax Check*

`Valid? Yes → continue / No → print error, skip`

↓ *Stage 4: Parser*

`CMD 1: argv=["echo", "/home/mmubina"], redirs=none`

`CMD 2: argv=["cat", "-e"], redirs=[REDIR_OUT → "out.txt"]`

Why not do it all in one step? Separation of concerns. Each stage has one job and does it well. The tokenizer doesn't care about variable names. The expander doesn't care about pipes. The parser doesn't care about raw characters. This makes each part simpler to write, test, and debug.

2. Enums (typedef enum)

What is it? An enum is a way to give names to integer values, making code readable. Instead of remembering "0 means word, 1 means pipe", you write TOKEN_WORD and TOKEN_PIPE.

```
typedef enum e_token_type
{
    TOKEN_WORD,        // = 0  (commands, arguments, filenames)
    TOKEN_PIPE,        // = 1  (the | operator)
    TOKEN_REDIR_IN,    // = 2  (the < operator)
    TOKEN_REDIR_OUT,   // = 3  (the > operator)
    TOKEN_HEREDOC,     // = 4  (the << operator)
    TOKEN_APPEND       // = 5  (the >> operator)
}   t_token_type;
```

How it works: C assigns integer values starting from 0, incrementing by 1. So `TOKEN_WORD == 0`, `TOKEN_PIPE == 1`, etc. You can use them in comparisons, switch statements, and function parameters.

Why use them? Compare these two lines:

```
if (token->type == 1)    // What does 1 mean? Who knows.
if (token->type == TOKEN_PIPE)  // Crystal clear.
```

3. Linked Lists

What is it? A chain of nodes where each node holds some data and a pointer to the next node. The last node points to NULL, marking the end of the list.

3.1 Why Not Arrays?

Arrays need a fixed size at creation time. But when tokenizing, we don't know how many tokens there will be until we finish scanning the input. Linked lists let us add nodes one at a time, growing dynamically.

3.2 Structure of a Node

```
typedef struct s_token
{
    char            *value;    // data: the token text (e.g., "ls")
    t_token_type    type;      // data: what kind of token
    struct s_token  *next;     // pointer to next node (or NULL)
}   t_token;
```

Visually, a linked list looks like this:

```
 [value:"ls"]     [value:"-la"]     [value:"|"]
 [type: WORD]     [type: WORD]      [type: PIPE]
 [next: ----] --> [next: ----] -->  [next: NULL]
```

3.3 The Three Linked Lists in This Project

Struct	Purpose	Where used
`t_token`	Chain of tokens from lexer output	tokenizer.c, token_utils.c
`t_cmd`	Chain of commands separated by pipes	parser.c, parser_utils.c
`t_redir`	Chain of redirections per command	parser.c, parser_utils.c

3.4 Core Operations

Create a node (new_token):

```
t_token  *new_token(char *value, t_token_type type)
{
    t_token  *token;

    token = malloc(sizeof(t_token));  // allocate memory for one node
    if (!token)
        return (NULL);                // malloc failed
    token->value = value;             // store the data
    token->type = type;
    token->next = NULL;               // not connected to anything yet
    return (token);
}
```

Key idea: `malloc(sizeof(t_token))` allocates exactly enough bytes for one `t_token` struct on the heap. We always check if malloc returns NULL (out of memory).

Append to end (add_token):

```
void  add_token(t_token **list, t_token *token)
{
    t_token  *tmp;

    if (!*list)                // if list is empty
    {
        *list = token;         // new node becomes the head
        return ;
    }
    tmp = *list;               // start at the head
    while (tmp->next)          // walk to the last node
        tmp = tmp->next;
    tmp->next = token;         // attach new node at the end
}
```

Key idea: To append, we must walk to the end. This is O(n) — we visit every node. For minishell inputs this is fine since commands are short.

Free entire list (free_tokens):

```
void  free_tokens(t_token *list)
{
    t_token  *tmp;
```

```
    while (list)
    {
        tmp = list->next;      // save the next pointer BEFORE freeing
        free(list->value);     // free the string data
        free(list);            // free the node itself
        list = tmp;            // move to next
    }
}
```

Critical: Always save `list->next` BEFORE calling `free(list)`. Once you free a node, its memory is gone — you can't access `list->next` after that.

4. Pointers to Pointers (t_token **list)

What is it? A pointer that stores the address of another pointer. Needed when a function must modify the caller's pointer (not just what it points to).

4.1 The Problem

In C, function arguments are passed by value — a copy. If you pass a pointer, the function gets a *copy* of that pointer. Modifying the copy doesn't change the original.

```
// THIS DOES NOT WORK:
void  add_token(t_token *list, t_token *token)
{
    if (!list)
        list = token;  // modifies the LOCAL copy, caller's pointer unchanged!
}

// In main:
t_token *tokens = NULL;
add_token(tokens, new_token("ls", TOKEN_WORD));
// tokens is STILL NULL here!
```

4.2 The Solution

Pass a pointer TO the pointer. Now the function can modify the original pointer via dereferencing:

```
// THIS WORKS:
void  add_token(t_token **list, t_token *token)
{
    if (!*list)
        *list = token;  // modifies what 'list' POINTS TO = the original pointer
}

// In main:
t_token *tokens = NULL;
add_token(&tokens, new_token("ls", TOKEN_WORD));
// tokens now points to the new node!
```

4.3 Visual Explanation

```
In main:          In add_token:

tokens ----       list (the **) points to tokens
  |                 |
  v                 v
 NULL        tokens (the caller's variable)
                    |
                    v
                  NULL

After *list = token:

tokens ----
  |
  v
 [WORD:"ls"] --> NULL
```

Rule of thumb: If a function needs to change *where a pointer points*, that function needs a pointer-to-pointer (`**`). If it only needs to read/modify the *data* the pointer points at, a single pointer (`*`) is enough.

5. State Machines (Quote Tracking)

What is it? A pattern where a variable tracks the current "state" of processing. The code behaves differently depending on what state it's in. In this project, the state is "are we inside quotes or not?"

This pattern appears in three places: `expand_string()`, `remove_quotes()`, and the tokenizer's `skip_quotes()`.

5.1 How Quote Tracking Works

```
char  quote = 0;           // state: 0 = not in quotes

while (str[i])
{
    if (!quote && (str[i] == '\'' || str[i] == '"'))
        quote = str[i];    // state: now inside this quote type
    else if (quote && str[i] == quote)
        quote = 0;         // state: closing quote found, back to normal
    // ... behavior depends on 'quote' state ...
    i++;
}
```

State transitions:

```
Input: echo "hello 'world'" done

State:  0  0  0  0  0  "  "  "  "  "  "  "  "  "  "  "  "  "  0  0  0  0  0
Char:   e  c  h  o     "  h  e  l  l  o     '  w  o  r  l  d  '  "     d  o  n  e

When quote = '"':
  - Single quotes inside are treated as regular characters
  - $ expansion still works (in expand_string)

When quote = '\'':
  - Double quotes inside are treated as regular characters
  - $ expansion is BLOCKED (everything is literal)
```

Why store the quote character and not just a boolean? Because single quotes and double quotes have different rules. By storing *which* quote opened, we know which character

closes it, and we know whether to expand variables.

6. Error Propagation with Return Values

What is it? A pattern where a function returns a special value (like -1 or NULL) to indicate failure, and every caller checks for that value and passes the error upward.

This is used in the tokenizer to handle unclosed quotes:

```
skip_quotes()      -- detects unclosed quote, returns -1
      |
      v
handle_word()      -- sees -1, returns -1 to its caller
      |
      v
tokenize()         -- sees -1, frees memory, returns NULL
      |
      v
main()             -- sees NULL, skips to next prompt
```

The chain in code:

```
// Level 1: detect the error
static int  skip_quotes(char *input, int i)
{
    ...
    if (!input[i])  // reached end without closing quote
    {
        write(2, "minishell: syntax error: unclosed quote\n", 40);
        return (-1);                    // signal error
    }
    return (i + 1);                     // signal success (new position)
}

// Level 2: check and propagate
static int  handle_word(char *input, int i, t_token **list)
{
    ...
    i = skip_quotes(input, i);
    if (i == -1)
        return (-1);                    // propagate error upward
    ...
}
```

```
// Level 3: clean up and propagate
t_token  *tokenize(char *input)
{
    ...
    i = handle_word(input, i, &list);
    if (i == -1)
    {
        free_tokens(list);              // clean up before returning
        return (NULL);                  // propagate as NULL
    }
    ...
}
```

Important: When propagating errors, always free any memory you've allocated before returning. In `tokenize()`, we call `free_tokens(list)` to free any tokens that were created before the error occurred. Otherwise: memory leak.

7. String Building Character by Character

What is it? Building a new string by appending one character (or one substring) at a time. Used in the expander when the output string has different length than the input (e.g., `$HOME` becomes `/home/mmubina`).

The pattern used in `expand_string()`:

```
result = ft_strdup("");                                  // start with empty string
while (str[i])
{
    if (/* this is a $VAR */)
        result = join_and_free(result, get_var_value(...));  // append expanded value
    else
        result = join_and_free(result, char_to_str(str[i++])); // append one char
}
```

Helper functions:

- `char_to_str(c)` — converts a single char to a malloc'd 2-byte string ("x\0")
- `join_and_free(s1, s2)` — joins two strings, frees both originals, returns new string

Why `join_and_free`? Each `ft_strjoin` creates a NEW string. The old `result` string is no longer needed. If we don't free it, that's a memory leak on every single character.

```
// Without join_and_free: MEMORY LEAK
result = ft_strjoin(result, "a");  // old result is lost, leaked!

// With join_and_free: CORRECT
result = join_and_free(result, char_to_str('a'));  // old result is freed
```

Part II — Code Walkthrough

8. Data Structures (minishell.h)

The header defines three structs that form the backbone of the parsing pipeline:

8.1 t_token — Output of the Lexer

```
typedef struct s_token
{
    char            *value;     // the text content, e.g., "ls", "|", ">"
    t_token_type    type;       // what kind: WORD, PIPE, REDIR_IN, etc.
    struct s_token  *next;      // next token in the linked list
}   t_token;
```

8.2 t_redir — A Single Redirection

```
typedef struct s_redir
{
    char            *file;      // target filename, e.g., "out.txt"
    t_token_type    type;       // which kind: REDIR_IN, REDIR_OUT, HEREDOC, APPEND
    struct s_redir  *next;      // next redirection (a command can have multiple)
}   t_redir;
```

8.3 t_cmd — Output of the Parser

```
typedef struct s_cmd
{
    char            **argv;         // NULL-terminated array: ["ls", "-la", NULL]
    t_redir         *redirections;  // linked list of redirections for this command
    struct s_cmd    *next;          // next command in the pipeline
}   t_cmd;
```

Relationship between them:

```
t_cmd 1                          t_cmd 2
+-------------------------+     +-------------------------+
| argv: ["cat", "f.txt"]  |     | argv: ["grep", "hello"] |
```

```
| redirections: NULL        |--->| redirections: ------+     |--> NULL
+---------------------------+    +---------------------|-----+
                                                       v
                                                  t_redir
                                                  +------------------+
                                                  | file: "out.txt"  |
                                                  | type: REDIR_OUT  |
                                                  | next: NULL       |
                                                  +------------------+
```

9. Tokenizer (tokenizer.c)

The tokenizer (also called "lexer") is the first stage. It scans the input string character by character and produces a linked list of tokens.

9.1 tokenize() — The Entry Point

```
t_token  *tokenize(char *input)
{
    t_token  *list;
    int      i;

    list = NULL;
    i = 0;
    while (input[i])
    {
        if (input[i] == ' ' || input[i] == '\t')
            i++;                                    // skip whitespace
        else if (is_operator_char(input[i]))
            i = handle_operator(input, i, &list);   // create operator token
        else
        {
            i = handle_word(input, i, &list);       // create word token
            if (i == -1)                            // unclosed quote detected
            {
                free_tokens(list);
                return (NULL);
            }
        }
    }
    return (list);
}
```

How it decides what to do:

Character	Action	Function called
Space or Tab	Skip (separator)	None, just i++
`|` `<` `>`	Create operator token	handle_operator()
Anything else	Start accumulating a word	handle_word()

9.2 is_operator_char()

```
static int  is_operator_char(char c)
{
    return (c == '|' || c == '<' || c == '>');
}
```

Simple helper: returns 1 (true) if the character is a shell operator, 0 otherwise.

9.3 handle_operator()

```
static int  handle_operator(char *input, int i, t_token **list)
{
    t_token_type  type;

    if (input[i] == '|')
    {
        add_token(list, new_token(ft_strdup("|"), TOKEN_PIPE));
        return (i + 1);          // consumed 1 character
    }
    type = get_redir_type(input, i);
    if (type == TOKEN_HEREDOC || type == TOKEN_APPEND)
    {
        add_token(list, new_token(ft_substr(input, i, 2), type));
        return (i + 2);          // consumed 2 characters (<< or >>)
    }
    add_token(list, new_token(ft_substr(input, i, 1), type));
    return (i + 1);              // consumed 1 character (< or >)
}
```

Logic: Pipe is always 1 char. For `<` and `>`, we check if the next char is the same to detect `<<` (heredoc) or `>>` (append). The function returns the new index position after consuming the operator character(s).

9.4 handle_word()

```
static int  handle_word(char *input, int i, t_token **list)
{
    int  start;

    start = i;
    while (input[i] && input[i] != ' ' && input[i] != '\t'
        && !is_operator_char(input[i]))
```

```
    {
        if (input[i] == '\'' || input[i] == '"')
        {
            i = skip_quotes(input, i);
            if (i == -1)
                return (-1);          // unclosed quote error
        }
        else
            i++;
    }
    add_token(list, new_token(ft_substr(input, start, i - start), TOKEN_WORD));
    return (i);
}
```

Logic: A "word" is everything until we hit whitespace or an operator. But if we encounter a quote, we call `skip_quotes()` to jump over the quoted section (keeping spaces inside quotes as part of the word). At the end, `ft_substr` extracts the substring from `start` to `i`.

Example: Input: `echo "hello world"`

```
handle_word called at i=5 (the 'e' after space... actually the '"')
Wait, let's trace from tokenize:

i=0: 'e' -> handle_word(input, 0, &list)
  start=0, walks e-c-h-o, hits ' ' at i=4
  creates [WORD:"echo"], returns 4

i=4: ' ' -> skip, i=5

i=5: '"' -> handle_word(input, 5, &list)
  start=5, input[5]='"' -> skip_quotes jumps to i=18 (past closing ")
  input[18] = '\0', loop ends
  creates [WORD:"\"hello world\""], returns 18
  (quotes are still in the value - removed later by expander)
```

9.5 skip_quotes()

```
static int  skip_quotes(char *input, int i)
{
    char  quote;

    quote = input[i];       // save which quote character (' or ")
    i++;                    // move past the opening quote
    while (input[i] && input[i] != quote)
        i++;                // scan until matching close or end of string
```

```
    if (!input[i])          // reached end without finding closing quote
    {
        write(2, "minishell: syntax error: unclosed quote\n", 40);
        return (-1);
    }
    return (i + 1);         // move past the closing quote
}
```

Error handling: If we reach `'\0'` without finding the closing quote, the quote is unclosed. We print an error and return -1, which propagates through `handle_word()` and `tokenize()` back to `main()` .

10. Token Utilities (token_utils.c)

This file contains the linked list operations for tokens (covered in Section 3.4) plus one additional function:

10.1 get_redir_type()

```
t_token_type  get_redir_type(char *str, int i)
{
    if (str[i] == '<')
    {
        if (str[i + 1] == '<')
            return (TOKEN_HEREDOC);      // <<
        return (TOKEN_REDIR_IN);         // <
    }
    if (str[i + 1] == '>')
        return (TOKEN_APPEND);           // >>
    return (TOKEN_REDIR_OUT);            // >
}
```

Decision tree:

```
Current char is '<'?
  ├── Next char is '<'?  --> TOKEN_HEREDOC   (<<)
  └── No                  --> TOKEN_REDIR_IN   (<)
Current char is '>'?
  ├── Next char is '>'?  --> TOKEN_APPEND    (>>)
  └── No                  --> TOKEN_REDIR_OUT (>)
```

10.2 token_list_size()

```
int  token_list_size(t_token *list)
{
    int  count;

    count = 0;
    while (list)
    {
        count++;
        list = list->next;
    }
```

```
    return (count);
}
```

Standard linked list length function — walk every node, count them.

11. Expander (expander.c + expander_utils.c)

The expander walks through each WORD token and replaces `$VAR` with its environment value and removes quotes. It uses the **state machine** (Section 5) and **string building** (Section 7) patterns.

11.1 expand_tokens() — Entry Point

```
void  expand_tokens(t_token *list, int exit_status)
{
    char  *expanded;
    char  *unquoted;

    while (list)
    {
        if (list->type == TOKEN_WORD)
        {
            expanded = expand_string(list->value, exit_status);
            free(list->value);           // free old value
            unquoted = remove_quotes(expanded);
            free(expanded);              // free intermediate value
            list->value = unquoted;      // store final value
        }
        list = list->next;
    }
}
```

Two-step process for each WORD token:

1. `expand_string()` — replaces `$VAR` with values (quotes still present)
2. `remove_quotes()` — strips the quote characters themselves

Order matters: we expand FIRST (because quotes affect expansion rules), then remove quotes.

11.2 expand_string()

```
char  *expand_string(char *str, int exit_status)
{
    char  *result;
    int   i;
```

```
    char  quote;

    result = ft_strdup("");
    i = 0;
    quote = 0;
    while (str[i])
    {
        if (!quote && (str[i] == '\'' || str[i] == '"'))
            quote = str[i];                // entering quotes
        else if (quote && str[i] == quote)
            quote = 0;                     // leaving quotes
        if (str[i] == '$' && quote != '\'' && str[i + 1]
            && str[i + 1] != ' ' && str[i + 1] != '"' && str[i + 1] != '\'')
            result = join_and_free(result, get_var_value(str, &i, exit_status));
        else
            result = join_and_free(result, char_to_str(str[i++]));
    }
    return (result);
}
```

Key rules:

- `$VAR` inside **double quotes** (`quote == '"'`) → expanded
- `$VAR` inside **single quotes** (`quote == '\''`) → NOT expanded (literal text)
- `$VAR` outside quotes → expanded

Example:

```
Input:  echo '$HOME' "$HOME"
                                    (assuming HOME=/home/mmubina)
Result: echo '$HOME' "/home/mmubina"
        (single-quoted part is untouched, double-quoted part is expanded)
```

11.3 get_var_value()

```
static char  *get_var_value(char *str, int *i, int exit_status)
{
    char  *name;
    char  *value;

    (*i)++;                             // skip the '$'
    if (str[*i] == '?')
    {
        (*i)++;
        return (ft_itoa(exit_status));   // $? = last exit code
```

```
    }
    if (!ft_isalpha(str[*i]) && str[*i] != '_')
        return (char_to_str('$'));       // bare $ with invalid next char
    name = get_var_name(str, i);         // extract variable name
    value = getenv(name);               // look up in environment
    free(name);
    if (value)
        return (ft_strdup(value));       // found: return its value
    return (ft_strdup(""));             // not found: return empty string
}
```

Note: The `i` parameter is `int *` (pointer), not `int`. This is because the function needs to advance the caller's index past the variable name. After `$HOME`, `i` should point past `E`, not stay at `$`.

11.4 Expander Utilities

join_and_free(): Joins two strings and frees both originals.

```
char  *join_and_free(char *s1, char *s2)
{
    char  *result;

    if (!s1) return (s2);
    if (!s2) return (s1);
    result = ft_strjoin(s1, s2);
    free(s1);
    free(s2);
    return (result);
}
```

char_to_str(): Converts one character to a malloc'd string.

```
char  *char_to_str(char c)
{
    char  *str;

    str = malloc(sizeof(char) * 2);   // 1 for char + 1 for '\0'
    if (!str) return (NULL);
    str[0] = c;
    str[1] = '\0';
    return (str);
}
```

remove_quotes(): Strips quote characters while respecting nesting.

```
char  *remove_quotes(char *str)
{
    char  *result;
    char  quote;
    int   i;
    int   j;

    result = malloc(sizeof(char) * (ft_len(str) + 1));
    if (!result) return (NULL);
    i = 0;
    j = 0;
    quote = 0;
    while (str[i])
    {
        if (!quote && (str[i] == '\'' || str[i] == '"'))
            quote = str[i];           // opening quote: skip this character
        else if (quote && str[i] == quote)
            quote = 0;               // closing quote: skip this character
        else
            result[j++] = str[i];    // normal char: copy it
        i++;
    }
    result[j] = '\0';
    return (result);
}
```

Example:

```
Input:   "hello"' world'
Process: " -> quote='"', skip
         h -> copy
         e -> copy
         l -> copy
         l -> copy
         o -> copy
         " -> quote=0, skip
         ' -> quote='\'', skip
           -> copy (space)
         w -> copy
         ... etc
         ' -> quote=0, skip
Result:  hello world
```

12. Syntax Checker (syntax_check.c)

After tokenizing and expanding, we validate the token stream for structural errors before parsing. This catches things like `| ls` or `cat >`.

12.1 check_syntax() — Entry Point

```
int  check_syntax(t_token *tokens)
{
    t_token  *tmp;
    int      is_first;

    tmp = tokens;
    is_first = 1;
    while (tmp)
    {
        if (tmp->type == TOKEN_PIPE && !check_pipe_syntax(tmp, is_first))
            return (0);            // error found
        if (is_redir_token(tmp->type) && !check_redir_syntax(tmp))
            return (0);            // error found
        is_first = 0;
        tmp = tmp->next;
    }
    return (1);                    // all good
}
```

Returns 1 for valid syntax, 0 for invalid. The `is_first` flag tracks whether we're looking at the first token (pipes can't be first).

12.2 Pipe Syntax Rules

```
static int  check_pipe_syntax(t_token *tok, int is_first)
{
    if (is_first)                              // pipe at start: | ls
    {
        write(2, "minishell: syntax error near '|'\n", 32);
        return (0);
    }
    if (!tok->next || tok->next->type == TOKEN_PIPE)  // pipe at end or ||
    {
        write(2, "minishell: syntax error near '|'\n", 32);
        return (0);
```

```
    }
    return (1);
}
```

Input	Error?	Why
`| ls`	Yes	Pipe is first token
`ls |`	Yes	Nothing after pipe (tok->next is NULL)
`ls || grep`	Yes	Next token is also a pipe
`ls | grep x`	No	Valid pipe usage

12.3 Redirection Syntax Rules

```
static int  check_redir_syntax(t_token *tok)
{
    if (!tok->next || tok->next->type != TOKEN_WORD)
    {
        write(2, "minishell: syntax error near redirection\n", 41);
        return (0);
    }
    return (1);
}
```

Input	Error?	Why
`cat >`	Yes	Nothing after redirection
`cat > | grep`	Yes	Next token is PIPE, not WORD
`cat > > file`	Yes	Next token is REDIR_OUT, not WORD
`cat > file`	No	Next token is WORD ("file")

13. Parser (parser.c + parser_utils.c)

The parser takes the validated token linked list and converts it into a linked list of `t_cmd` structs — the format that the execution engine needs.

13.1 parse_tokens() — Entry Point

```
t_cmd  *parse_tokens(t_token *tokens)
{
    t_cmd   *list;
    t_cmd   *cmd;
    t_token *current;

    list = NULL;
    current = tokens;
    while (current)
    {
        cmd = new_cmd();
        cmd->argv = build_argv(current, count_args(current));
        extract_redirs(current, cmd);
        add_cmd(&list, cmd);
        current = skip_to_pipe(current);   // jump to next command
    }
    return (list);
}
```

For each segment between pipes:

1. Create a new empty command
2. Count WORD tokens and build argv array
3. Extract any redirections
4. Add command to the list
5. Skip past the pipe to the next segment

13.2 count_args()

```
static int  count_args(t_token *tokens)
{
    int  count;

    count = 0;
```

```
    while (tokens && tokens->type != TOKEN_PIPE)
    {
        if (tokens->type == TOKEN_WORD)
            count++;                    // count this word
        else if (tokens->type != TOKEN_PIPE)
            tokens = tokens->next;      // skip the redir's filename
        tokens = tokens->next;
    }
    return (count);
}
```

Why skip after a redirection? In `cat > file.txt -n` , the tokens are `[cat] [>] [file.txt] [-n]` . The `file.txt` after `>` is the redirection target, NOT an argument to cat. So when we see a non-WORD non-PIPE token (a redirection), we skip the next token (the filename) to avoid counting it as an argument.

13.3 build_argv()

```
static char  **build_argv(t_token *tokens, int count)
{
    char  **argv;
    int   i;

    argv = malloc(sizeof(char *) * (count + 1));   // +1 for NULL terminator
    if (!argv) return (NULL);
    i = 0;
    while (tokens && tokens->type != TOKEN_PIPE)
    {
        if (tokens->type == TOKEN_WORD)
            argv[i++] = ft_strdup(tokens->value);  // copy word into argv
        else if (tokens->type != TOKEN_PIPE)
            tokens = tokens->next;                 // skip redir filename
        tokens = tokens->next;
    }
    argv[i] = NULL;                                // NULL-terminate (required by exec
    return (argv);
}
```

Why NULL-terminate? The `execve()` system call requires argv to end with a NULL pointer. This is how it knows where the argument list ends.

13.4 extract_redirs()

```
static void  extract_redirs(t_token *tokens, t_cmd *cmd)
{
    while (tokens && tokens->type != TOKEN_PIPE)
    {
        if (tokens->type != TOKEN_WORD && tokens->type != TOKEN_PIPE
            && tokens->next)
        {
            add_redir(&cmd->redirections,
                new_redir(ft_strdup(tokens->next->value), tokens->type));
        }
        tokens = tokens->next;
    }
}
```

Walks the tokens looking for redirection operators. When found, saves the operator type and the next token's value (the filename) as a `t_redir` node.

13.5 skip_to_pipe()

```
static t_token  *skip_to_pipe(t_token *tokens)
{
    while (tokens && tokens->type != TOKEN_PIPE)
        tokens = tokens->next;      // walk past all non-pipe tokens
    if (tokens)
        tokens = tokens->next;      // skip the pipe token itself
    return (tokens);                // return first token of next command (or NULL)
}
```

13.6 Parser Utilities (parser_utils.c)

Contains `new_cmd()`, `add_cmd()`, `new_redir()`, `add_redir()` — same linked list patterns as token_utils.c.

free_cmds() deserves attention because it frees three levels:

```
void  free_cmds(t_cmd *list)
{
    t_cmd   *tmp;
    t_redir *rtmp;
    int     i;

    while (list)
    {
        tmp = list->next;
```

```
        if (list->argv)                         // 1. Free argv array
        {
            i = 0;
            while (list->argv[i])
                free(list->argv[i++]);          //    Free each string
            free(list->argv);                   //    Free the array itself
        }
        while (list->redirections)              // 2. Free redirection list
        {
            rtmp = list->redirections->next;
            free(list->redirections->file);     //    Free filename string
            free(list->redirections);           //    Free redir node
            list->redirections = rtmp;
        }
        free(list);                             // 3. Free the cmd node
        list = tmp;
    }
}
```

14. Main Loop (main.c)

```
int  main(int argc, char **argv, char **envp)
{
    char    *input;
    t_token *tokens;
    t_cmd   *cmds;

    (void)argc;
    (void)argv;
    (void)envp;
    while (1)
    {
        input = readline("minishell> ");    // 1. Read user input
        if (!input)
            break ;                         // Ctrl+D: exit
        if (input[0] != '\0')
            add_history(input);             // 2. Save to history
        tokens = tokenize(input);           // 3. Tokenize
        free(input);
        if (!tokens)
            continue ;                      // Empty or error: next prompt
        expand_tokens(tokens, 0);           // 4. Expand variables
        if (!check_syntax(tokens))          // 5. Validate syntax
        {
            free_tokens(tokens);
            continue ;                      // Syntax error: next prompt
        }
        cmds = parse_tokens(tokens);        // 6. Parse into commands
        print_cmds(cmds);                   // 7. (Debug) Print result
        free_tokens(tokens);                // 8. Clean up
        free_cmds(cmds);
    }
    return (0);
}
```

Memory management flow:

- `input` — allocated by `readline()`, freed after tokenizing
- `tokens` — allocated by `tokenize()`, freed after parsing
- `cmds` — allocated by `parse_tokens()`, freed after execution (currently printing)

Part III — Full Example Trace

15. End-to-End Walkthrough

Let's trace the complete input: `cat -n < in.txt | grep "hello $USER" >> log.txt`

Stage 1: Tokenizer

```
i=0  'c' -> handle_word  -> [WORD:"cat"]       i=3
i=3  ' ' -> skip                                i=4
i=4  '-' -> handle_word  -> [WORD:"-n"]         i=6
i=6  ' ' -> skip                                i=7
i=7  '<' -> handle_operator -> [REDIR_IN:"<"]   i=8
i=8  ' ' -> skip                                i=9
i=9  'i' -> handle_word  -> [WORD:"in.txt"]     i=15
i=15 ' ' -> skip                                i=16
i=16 '|' -> handle_operator -> [PIPE:"|"]       i=17
i=17 ' ' -> skip                                i=18
i=18 'g' -> handle_word  -> [WORD:"grep"]       i=22
i=22 ' ' -> skip                                i=23
i=23 '"' -> handle_word:
             skip_quotes jumps to i=36 (past closing ")
             -> [WORD:"\"hello $USER\""]        i=36
i=36 ' ' -> skip                                i=37
i=37 '>' -> handle_operator:
             next char is '>' -> TOKEN_APPEND
             -> [APPEND:">>"]                    i=39
i=39 ' ' -> skip                                i=40
i=40 'l' -> handle_word  -> [WORD:"log.txt"]    i=47
```

Token list:

```
[WORD:"cat"] -> [WORD:"-n"] -> [REDIR_IN:"<"] -> [WORD:"in.txt"] -> [PIPE:"|"]
  -> [WORD:"grep"] -> [WORD:"\"hello $USER\""] -> [APPEND:">>"] -> [WORD:"log.txt"]
```

Stage 2: Expander

```
Only TOKEN_WORD tokens are processed:
```

```
"cat"                -> "cat"                (no $ or quotes)
"-n"                 -> "-n"                 (no $ or quotes)
"in.txt"             -> "in.txt"             (no $ or quotes)
"\"hello $USER\""    -> "\"hello mmubina\"" (expand $USER inside double quotes)
  then remove_quotes -> "hello mmubina"      (strip the double quotes)
"log.txt"            -> "log.txt"            (no $ or quotes)
```

Stage 3: Syntax Check

```
[WORD] - ok (is_first=1, not pipe/redir)
[WORD] - ok
[REDIR_IN] - check_redir_syntax: next is [WORD:"in.txt"] -> ok
[WORD] - ok
[PIPE] - check_pipe_syntax: not first, next is [WORD:"grep"] -> ok
[WORD] - ok
[WORD] - ok
[APPEND] - check_redir_syntax: next is [WORD:"log.txt"] -> ok
[WORD] - ok
Result: 1 (valid)
```

Stage 4: Parser

```
Segment 1 (before pipe): [WORD:"cat"] [WORD:"-n"] [REDIR_IN] [WORD:"in.txt"]
  count_args: "cat"=1, "-n"=2, REDIR_IN=skip next, result=2
  build_argv: ["cat", "-n", NULL]
  extract_redirs: REDIR_IN with file "in.txt"
  -> CMD 1: argv=["cat", "-n"], redirections=[REDIR_IN -> "in.txt"]

Segment 2 (after pipe): [WORD:"grep"] [WORD:"hello mmubina"] [APPEND] [WORD:"log.txt"]
  count_args: "grep"=1, "hello mmubina"=2, APPEND=skip next, result=2
  build_argv: ["grep", "hello mmubina", NULL]
  extract_redirs: APPEND with file "log.txt"
  -> CMD 2: argv=["grep", "hello mmubina"], redirections=[APPEND -> "log.txt"]
```

Final result:

```
t_cmd 1                                  t_cmd 2
+------------------------------+         +------------------------------+
| argv: ["cat", "-n"]          |         | argv: ["grep","hello mmubina"]|
| redirections:                 |    -->   | redirections:                 |  --> NULL
```

```
|   [REDIR_IN -> "in.txt"]    |          |   [APPEND -> "log.txt"]      |
+-----------------------------+          +------------------------------+
```

This is exactly what the execution engine needs to run the pipeline.

Quick Reference: All Files

File	Purpose	Key functions
minishell.h	Data structures + prototypes	t_token, t_cmd, t_redir, t_token_type
tokenizer.c	Lexer: string to tokens	tokenize(), handle_word(), handle_operator(), skip_quotes()
token_utils.c	Token linked list ops	new_token(), add_token(), free_tokens(), get_redir_type()
expander.c	Variable expansion	expand_tokens(), expand_string(), get_var_value()
expander_utils.c	String helpers	join_and_free(), char_to_str(), remove_quotes()
syntax_check.c	Token validation	check_syntax(), check_pipe_syntax(), check_redir_syntax()
parser.c	Tokens to commands	parse_tokens(), build_argv(), extract_redirs(), count_args()
parser_utils.c	Command linked list ops	new_cmd(), add_cmd(), free_cmds(), new_redir(), add_redir()
main.c	Shell loop	main(), print_cmds()