🔧 Minishell - Teknik Analiz ve Kod Yapısı

📋 İçindekiler

1. Veri Yapıları Detayı
2. İşlev Akış Diyagramları
3. Kritik Kod Segmentleri
4. Hata Yönetimi Stratejisi
5. Performans Optimizasyonları
6. Test Senaryoları

🗂️ Veri Yapıları Detayı

📦 Temel Yapılar

`t_req` - İstek Yapısı (Ana Kontrol)

```
typedef struct s_request {
    t_list    *cmds;        // Komutların bağlı listesi
    char      **envp;       // Ortam değişkenleri dizisi
    pid_t     pid;          // Ana süreç ID'si
    int       exit_stat;    // Son komutun çıkış durumu
    int       should_exit;  // Çıkış bayrağı (global yerine)
} t_req;

// Kullanım örneği:
t_req req = setup(av, env);
req.should_exit = 0;  // Çalışmaya devam et
req.exit_stat = 0;    // Başarılı durum
```

`t_shell` - Komut Yapısı

```
typedef struct s_shell {
    char        **full_cmd;     // ["ls", "-la", "/home", NULL]
    char        *full_path;     // "/usr/bin/ls"
    int         infile;         // Girdi FD (varsayılan: STDIN_FILENO)
    int         outfile;        // Çıktı FD (varsayılan: STDOUT_FILENO)
    char        *infile_path;   // Girdi dosya yolu
    char        *outfile_path;  // Çıktı dosya yolu
    int         append_out;     // Eklemeli mod bayrağı
    t_redirect  *redirects;     // Yönlendirme zinciri
} t_shell;

// Bellek yerleşimi örneği:
t_shell cmd = {
```

```
    .full_cmd = ["echo", "hello", NULL],
    .full_path = "/usr/bin/echo",
    .infile = 0,    // STDIN
    .outfile = 1,   // STDOUT
    .redirects = NULL
};
```

`t_token` - **Token (Parçacık) Yapısı**

```
typedef struct s_token {
    char            *str;    // Token (parçacık) metni
    t_quote_type    quote;   // QUOTE_NONE/QUOTE_SINGLE/QUOTE_DOUBLE
} t_token;

// Token dizi örneği:
t_token **tokens = {
    {"echo", QUOTE_NONE},
    {"'hello world'", QUOTE_SINGLE},
    {"|", QUOTE_NONE},
    {"grep", QUOTE_NONE},
    {"hello", QUOTE_NONE},
    {NULL, QUOTE_NONE}  // Sonlandırıcı
};
```

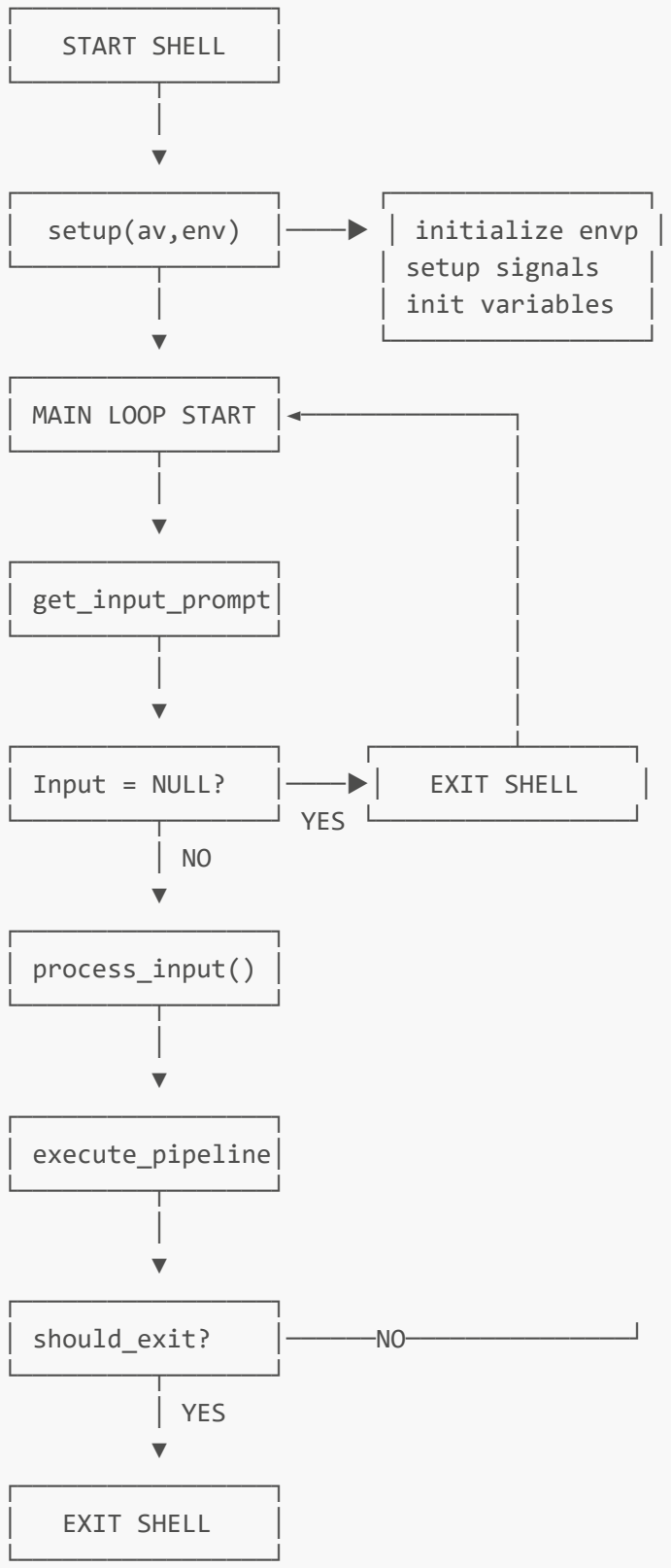
Bağlı Liste (Linked List) Kullanımı

```
// Komut zinciri yapısı:
t_list *cmds = NULL;
ft_lstadd_back(&cmds, ft_lstnew(cmd1));  // echo hello
ft_lstadd_back(&cmds, ft_lstnew(cmd2));  // | grep hello
ft_lstadd_back(&cmds, ft_lstnew(cmd3));  // > output.txt

// Dolaşım (Traversal):
t_list *current = cmds;
while (current) {
    t_shell *cmd = (t_shell *)current->content;
    execute_command(cmd);
    current = current->next;
}
```

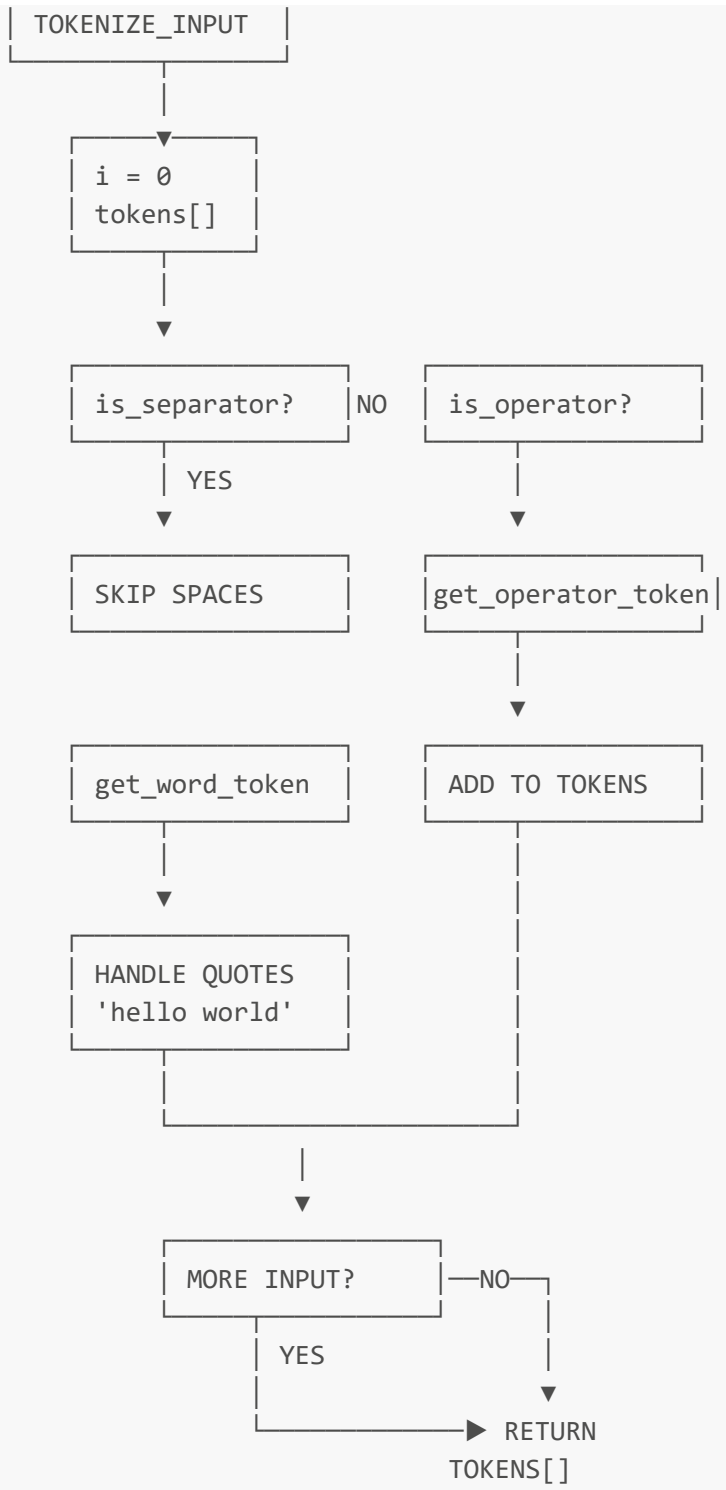
İşlev Akış Diyagramları

Ana Çalışma Akışı

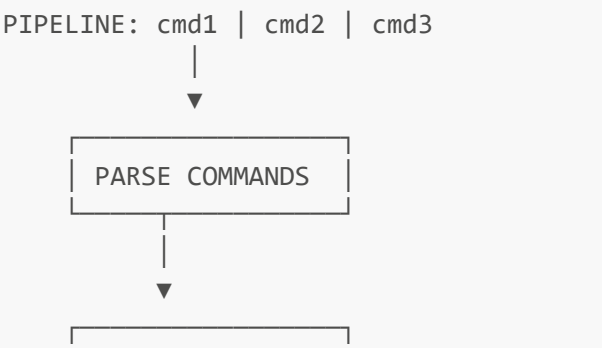


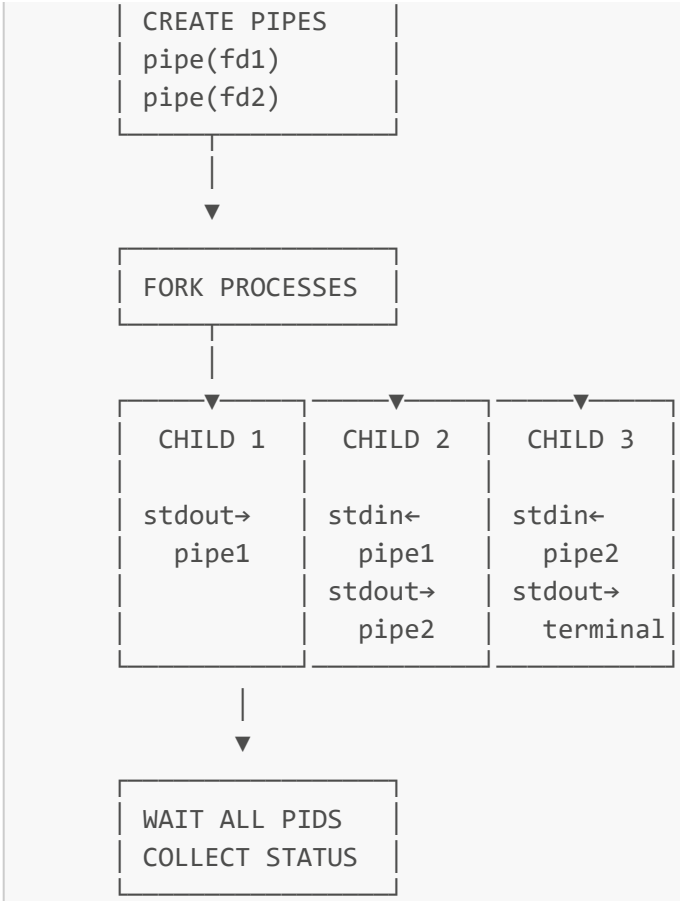
Tokenizasyon Süreci

```
INPUT: "echo 'hello world' | grep hello > output.txt"
          │
          ▼
┌─────────────────┐
```



Pipeline (Boru Hattı) Yürütme Akışı





🔧 Kritik Kod Segmentleri

◇ Tokenizer Temel Mantığı

```
t_token *get_word_token(const char *input, int *i) { // tokenizer_core.c - get_word_token()
t_token *get_word_token(const char *input, int *i) {
    int has_single = 0, has_double = 0, has_unquoted = 0;
    char *result = malloc(32);  // Initial capacity
    t_token_state state = {0, 32};

    while (input[*i] && !is_separator(input[*i]) && !is_operator(input[*i])) {
        if (input[*i] == '\'' || input[*i] == '"') {
            // ÖNEMLİ: Tırnak işareti yönetimi
            if (input[*i] == '\'') has_single = 1;
            else has_double = 1;

            if (!handle_quoted_section(input, i, &result, &state))
                return (free(result), NULL);
        } else {
            has_unquoted = 1;
            if (!append_char_to_result(&result, &state.len, &state.capacity, input[*i]))
                return (free(result), NULL);
            (*i)++;
```

```
        }
    }

    return create_token_and_free(result,
           determine_quote_type(has_single, has_double, has_unquoted));
}

// ÖNEMLİ: Tırnak tipi belirleme algoritması
int determine_quote_type(int has_single, int has_double, int has_unquoted) {
    if (has_single && !has_double && !has_unquoted) return QUOTE_SINGLE;
    if (has_double && !has_single && !has_unquoted) return QUOTE_DOUBLE;
    return QUOTE_NONE;  // Mixed veya unquoted
}
```

◇ Değişken Genişletme Motoru

```
// expander_process.c - process_input_loop()
char *process_input_loop(const char *input, char **envp, t_req *req) {
    int i = 0, len = 1;
    char *result = initialize_result();

    while (input[i]) {
        if (input[i] == '$') {
            // ÖNEMLİ: Değişken genişletme
            char *expanded = process_variable(input, &i, envp, req);
            if (!expanded) return (free(result), NULL);

            result = append_str(result, expanded, &len);
            free(expanded);
        } else {
        // Normal karakter
            result = process_character(result, input, &i, &len);
        }
    }
    return result;
}

// Değişken çıkarma mantığı
char *expand_var(const char *input, int *i, char **envp, t_req *req) {
    (*i)++; // Skip '$'

    if (input[*i] == '?') {
        (*i)++;
        return ft_itoa(req->exit_stat);  // Çıkış durumu genişletme
    }

    // Değişken adını çıkar
    int start = *i;
    while (input[*i] && (ft_isalnum(input[*i]) || input[*i] == '_'))
        (*i)++;
```

```
    char *var_name = ft_substr(input, start, *i - start);
    char *value = mini_getenv(var_name, envp, ft_strlen(var_name));
    free(var_name);

    return value ? ft_strdup(value) : ft_strdup("");
}
```

◇ Pipeline (Boru Hattı) Süreç Yönetimi

```
// executor_pipeline.c - execute_loop()
static void execute_loop(t_list *cmds, pid_t *pids, int count, t_req *req) {
    t_pipeline_data data = {
        .input_fd = STDIN_FILENO,
        .i = 0,
        .pids = pids
    };

    t_list *node = cmds;
    while (node) {
        t_shell *cmd = (t_shell *)node->content;

        // ÖNEMLİ: Tek yerleşik komut optimizasyonu
        if (count == 1 && cmd->full_cmd && is_builtin(cmd->full_cmd[0])) {
            handle_single_builtin(cmds, req, pids, data.i);
        } else {
            execute_single_cmd(node, cmd, &data, req);
        }

        data.i++;
        node = node->next;
    }

    wait_for_processes(pids, count, req);
    if (data.input_fd != STDIN_FILENO)
        close(data.input_fd);
}

// Process fork and execution
pid_t exec_external_cmd(t_shell *cmd, t_req *req, int in_fd, int out_fd) {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        return -1;
    }

    if (pid == 0) {
        // ÇOCUK SÜREÇ
        close_extra_fds(in_fd, out_fd);
        reset_signals();
```

```
        // Dosya tanımlayıcılarını ayarla
        if (apply_redirects(cmd, req)) exit(1);

        if (cmd->infile != STDIN_FILENO)
            set_fd(cmd->infile, STDIN_FILENO);
        else
            set_fd(in_fd, STDIN_FILENO);

        if (cmd->outfile != STDOUT_FILENO)
            set_fd(cmd->outfile, STDOUT_FILENO);
        else
            set_fd(out_fd, STDOUT_FILENO);

        // Komutu çalıştır
        execve(cmd->full_path, cmd->full_cmd, req->envp);
        handle_execve_error(cmd, req);
        exit(req->exit_stat);
    }

    return pid;  // PARENT returns child PID
}
```

◇ Heredoc (Çok Satırlı Girdi) Uygulaması

```
// heredoc_handler.c - handle_heredoc()
int handle_heredoc(const char *delimiter, t_req *req) {
    int pipe_fd[2];
    pid_t pid;
    void (*old_sigint)(int);

    // ÖNEMLİ: Sinyal maskeleme
    old_sigint = signal(SIGINT, SIG_IGN);

    if (pipe(pipe_fd) == -1) {
        perror("minishell: pipe");
        signal(SIGINT, old_sigint);
        return -1;
    }

    pid = fork();
    if (pid == 0) {
        // ÇOCUK: Heredoc girdi toplama
        signal(SIGINT, heredoc_sigint_handler);
        do_heredoc_child(delimiter, pipe_fd);
    }

    // EBEVEYN: Çocuğu bekle
    close(pipe_fd[1]);
    waitpid(pid, &status, 0);
    signal(SIGINT, old_sigint);
```

```
    // ÖNEMLİ: Sinyal yönetimi kontrolü
    if (WIFSIGNALED(status) && WTERMSIG(status) == SIGINT) {
        req->exit_stat = 130;  // Bash-compatible
        close(pipe_fd[0]);
        return -1;
    }

    return pipe_fd[0];  // Pipe'ın okuma ucunu döndür
}

// Child process logic
static int do_heredoc_child(const char *delimiter, int pipe_fd[2]) {
    char *line;

    signal(SIGINT, heredoc_sigint_handler);
    close(pipe_fd[0]);

    while (1) {
        line = readline("> ");
        if (!line) {
            // ÖNEMLİ: Ctrl+C algılama
            close(pipe_fd[1]);
            _exit(130);
        }

        if (!ft_strncmp(line, delimiter, ft_strlen(delimiter) + 1)) {
            free(line);
            break;
        }

        write(pipe_fd[1], line, ft_strlen(line));
        write(pipe_fd[1], "\n", 1);
        free(line);
    }

    close(pipe_fd[1]);
    exit(0);
}
```

🚨 Hata Yönetimi Stratejisi

◇ Bellek Yönetimi

```
// Pattern: RAII (Resource Acquisition Is Initialization)
t_token **tokenize_input(const char *input) {
    t_token **tokens = malloc(sizeof(t_token *) * capacity);
    if (!tokens) return NULL;

    // ... processing ...
```

```
    // Error case:
    if (error_occurred) {
        free_tokens(tokens);  // Cleanup before return
        return NULL;
    }

    return tokens;  // Success
}

// Cleanup fonksiyonları:
void free_tokens(t_token **tokens) {
    if (!tokens) return;

    for (int i = 0; tokens[i]; i++) {
        free(tokens[i]->str);
        free(tokens[i]);
    }
    free(tokens);
}
// Desen: RAII (Kaynak Edinimi Başlatma Anında)
// Temizleme fonksiyonları:
// Bellek yönetimi
// Hata yönetimi
// İlgili açıklamalar
```

◇ Sistem Çağrısı Hata Yönetimi

```
// execve hata sınıflandırması:
static void handle_execve_error(t_shell *cmd, t_req *req) {
    if (errno == EISDIR)
        ms_error(ERR_IS_DIR, cmd->full_path, 126, req);
    else if (errno == EACCES) {
        if (access(cmd->full_path, X_OK) == 0)
            ms_error(ERR_IS_DIR, cmd->full_path, 126, req);
        else
            ms_error(ERR_NO_PERM, cmd->full_path, 126, req);
    }
    else if (errno == ENOENT)
        ms_error(ERR_NO_CMD, cmd->full_path, 127, req);
    else {
        perror("execve");
        req->exit_stat = 1;
    }
}

// Çıkış kodu standartları (bash uyumlu):
// 0    - Başarılı
// 1    - Genel hata
// 2    - Söz dizimi hatası
// 126  - İzin yok
```

```
// 127  - Komut bulunamadı
// 130  - Ctrl+C (SIGINT)
```

Performans Optimizasyonları

◇ Bellek Havuzu (Memory Pool) Deseni

```
// Token dizisinin dinamik yeniden boyutlanması:
int resize_token_array(t_token ***tokens, int *capacity, int count) {
    int new_capacity = (*capacity) * 2;
    t_token **new_tokens = realloc(*tokens, sizeof(t_token *) * new_capacity);

    if (!new_tokens) return 0;

    *tokens = new_tokens;
    *capacity = new_capacity;
    return 1;
}

// String oluşturma optimizasyonu:
static char *reallocate_result(char *result, int *capacity) {
    int new_capacity = (*capacity) * 2;  // Exponential growth
    char *new_result = malloc(new_capacity);

    if (!new_result) {
        free(result);
        return NULL;
    }

    ft_strlcpy(new_result, result, *capacity);
    free(result);
    *capacity = new_capacity;
    return new_result;
}
```

◇ Süreç (Process) Optimizasyonu

```
// Tek yerleşik komut optimizasyonu (fork gerekmez):
if (count == 1 && cmd->full_cmd && is_builtin(cmd->full_cmd[0])) {
    exec_single_builtin(cmd, req, input_fd);
    return;  // No fork/wait overhead
}

// Pipe kurulumunu minimumda tutma:
if (node->next) {
    // Only create pipe if there's a next command
    if (pipe(pipe_fd) == -1) return -1;
```

```
    output_fd = pipe_fd[1];
}
```

Test Senaryoları

Birim Test Örnekleri

```
# Test 1: Temel Komut
$ echo hello
Expected: "hello\n"
Exit: 0

# Test 2: Boru Hattı (Pipeline)
$ echo hello | grep hello
Expected: "hello\n"
Exit: 0

# Test 3: Yönlendirme
$ echo hello > /tmp/test && cat /tmp/test
Expected: "hello\n"
Exit: 0

# Test 4: Değişken Genişletme
$ export TEST=world && echo hello $TEST
Expected: "hello world\n"
Exit: 0

# Test 5: Tırnak Yönetimi
$ echo 'hello world' "test $USER"
Expected: "hello world test username\n"
Exit: 0

# Test 6: Hata Durumları
$ /nonexistent/command
Expected: stderr message
Exit: 127

$ echo hello |
Expected: syntax error message
Exit: 2

# Test 7: Sinyal Yönetimi
$ cat << EOF
> line1
> ^C
Expected: interrupt, return to prompt
Exit: 130

# Test 8: Karmaşık Boru Hattı
$ env | grep HOME | cut -d= -f2
```

```
Expected: "/Users/username\n"
Exit: 0
```

◇ Bellek Sızıntısı Testi

```
# Valgrind ile test:
$ valgrind --leak-check=full --show-leak-kinds=all ./minishell

# Beklenen çıktı:
# ==PID== HEAP SUMMARY:
# ==PID==     in use at exit: 0 bytes in 0 blocks
# ==PID==   total heap usage: N allocs, N frees, X bytes allocated
# ==PID==
# ==PID== All heap blocks were freed -- no leaks are possible

# Test komutları:
(minishell) $ echo hello | grep hello > /tmp/test
(minishell) $ cat /tmp/test
(minishell) $ export TEST=value && echo $TEST
(minishell) $ cd /tmp && pwd
(minishell) $ exit
```

◇ Stres Testi

```
# Uzun boru hattı testi:
$ cat /etc/passwd | grep user | cut -d: -f1 | sort | uniq | head -10

# Çoklu yönlendirme:
$ echo test > file1 && cat file1 > file2 && cat file2

# İç içe tırnaklar:
$ echo "outer 'inner' quote" 'outer "inner" quote'

# Büyük heredoc:
$ cat << EOF
> line 1
> line 2
> ... (many lines)
> line N
> EOF

# Arka plan süreç simülasyonu:
$ sleep 5 &  # Not implemented, should error appropriately
```

📊 Kod Metrikleri

◇ Fonksiyon Karmaşıklığı

```
Tokenizer Modülü:    ~15 fonksiyon, ortalama karmaşıklık: Orta
Parser Modülü:       ~12 fonksiyon, ortalama karmaşıklık: Yüksek
Executor Modülü:     ~20 fonksiyon, ortalama karmaşıklık: Yüksek
Expander Modülü:     ~8 fonksiyon,  ortalama karmaşıklık: Orta
Builtin Modülü:      ~15 fonksiyon, ortalama karmaşıklık: Düşük
```

◇ Bellek Kullanım Desenleri

```
Token Dizisi:     Dinamik (16 → 32 → 64 → ...)
Komut Listesi:    Bağlı Liste (O(n) dolaşım)
Ortam:            Statik Dizi (char **)
Borular:          Yığında ayrılmış çiftler
Process ID'ler:   Dinamik dizi (pid_t *)
```

◇ Zaman Karmaşıklığı

```
Tokenizasyon:    O(n) (n = girdi uzunluğu)
Parse:           O(t) (t = token sayısı)
Genişletme:      O(n*v) (v = değişken sayısı)
Çalıştırma:      O(c) (c = komut sayısı)
Boru Hattı:      O(c) paralel yürütme
```

Teknik Dokümantasyon Tarihi: 27 Temmuz 2025 Detay Seviyesi: İleri Düzey Hedef Kitle: Geliştiriciler & Kod İnceleyiciler