Документация к файлу parcer.c
Парсер командной строки

Разработчик: [Ваше имя]

16 ноября 2025 г.

Аннотация

Данный документ описывает структуру и функциональность модуля парсинга командной строки, реализованного в файле `parcer.c`. Модуль отвечает за преобразование текстового ввода пользователя в структурированные команды с учетом специальных символов, кавычек, экранирования и перенаправлений.

Содержание

1 Обзор модуля

Файл `parcer.c` реализует сложный лексический анализатор командной строки, который обрабатывает:

- Разделение на токены с учетом пробельных символов
- Обработку двойных кавычек и экранирования
- Распознавание специальных символов и операторов
- Обработку перенаправлений ввода/вывода
- Парсинг конвейеров и последовательностей команд
- Выделение фонового режима выполнения

2 Структура модуля

2.1 Заголовочные файлы

```c
#include "shell.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

2.2 Глобальные константы и переменные

```c
static const char DELIMITERS[] = " \t\n\r";
```

2.3 Прототипы внутренних функций

```
void process_redirections_and_pipes(command_t *cmd, char **words, int *
    word_num);
```

3 Вспомогательные функции

3.1 Функция is_special_char

```
int is_special_char(char c) {
    const char *special_chars = "@#%!&$^;:,(){}[]";
    return (strchr(special_chars, c) != NULL);
}
```

Назначение: Проверка, является ли символ специальным.
Параметры:

- c - проверяемый символ

Возвращаемое значение: 1 если символ специальный, 0 в противном случае.

3.2 Функция is_redirection_char

```
int is_redirection_char(char *str) {
    return (strcmp(str, ">") == 0 || strcmp(str, ">>") == 0 ||
            strcmp(str, "<") == 0 || strcmp(str, "2>") == 0 ||
            strcmp(str, "2>>") == 0 || strcmp(str, "2>&1") == 0);
}
```

Назначение: Проверка, является ли строка оператором перенаправления.

3.3 Функция is_command_separator

```
int is_command_separator(const char *str) {
    return (strcmp(str, ";") == 0 || strcmp(str, "&&") == 0 || strcmp(str, "
    ||") == 0);
}
```

Назначение: Проверка, является ли строка разделителем команд.

3.4 Функция get_separator_type

```
int get_separator_type(const char *sep) {
    if (strcmp(sep, ";") == 0) return 0;
    if (strcmp(sep, "&&") == 0) return 1;
    if (strcmp(sep, "||") == 0) return 2;
    return -1;
}
```

Назначение: Получение числового типа разделителя.
Коды возврата:

- 0 - точка с запятой (;)

- 1 - логическое И (&&)
- 2 - логическое ИЛИ (||)
- -1 - неизвестный разделитель

4 Основные функции парсинга

4.1 Функция `parse_input`

```
command_t *parse_input(const char *input) {
    if (input == NULL || strlen(input) == 0) {
        return NULL;
    }

    command_t *cmd = malloc(sizeof(command_t));
    if (cmd == NULL) {
        perror("malloc");
        return NULL;
    }

    //

    cmd->words = NULL;
    cmd->word_num = 0;
    cmd->fonius = 0;
    cmd->input_file = NULL;
    cmd->output_file = NULL;
    cmd->error_file = NULL;
    cmd->append_output = 0;
    cmd->append_error = 0;
    cmd->merge_output = 0;
    cmd->pipeline = NULL;
    cmd->pipeline_count = 0;

    char *line = strdup(input);
    if (line == NULL) {
        perror("strdup");
        free(cmd);
        return NULL;
    }
    // ...
}
```

Назначение: Основная функция парсинга входной строки в структуру команды.
Алгоритм работы:

1. Проверка входных данных и выделение памяти
2. Инициализация структуры `command_t`
3. Копирование входной строки для обработки
4. Посимвольный анализ с учетом:
    - Экранирования обратным слешем
    - Двойных кавычек

- Специальных символов и операторов

5. Обработка фонового режима
6. Обработка перенаправлений и конвейеров
7. Формирование итогового массива слов

Особенности обработки токенов:

- **Двойные символы**: `»`, `«`, `&&`, `||` обрабатываются как единые токены
- **Перенаправления stderr**: `2>`, `2»`, `2>&1` выделяются отдельно
- **Кавычки**: Текст внутри двойных кавычек обрабатывается как единое целое
- **Экранирование**: Символ \ экранирует следующий символ

4.2 Функция `remove_words`

```
void remove_words(char **words, int *count, int start, int num) {
    if (start < 0 || start >= *count || num <= 0) return;

    //
    for (int i = start; i < start + num && i < *count; i++) {
        free(words[i]);
    }

    //
    for (int i = start; i < *count - num; i++) {
        words[i] = words[i + num];
    }

    *count -= num;
}
```

Назначение: Удаление группы токенов из массива слов.

4.3 Функция `copy_redirections`

```
void copy_redirections(command_t *dest, const command_t *src) {
    if (src->input_file) dest->input_file = strdup(src->input_file);
    if (src->output_file) dest->output_file = strdup(src->output_file);
    if (src->error_file) dest->error_file = strdup(src->error_file);
    dest->append_output = src->append_output;
    dest->append_error = src->append_error;
    dest->merge_output = src->merge_output;
    dest->fonius = src->fonius;
}
```

Назначение: Копирование информации о перенаправлениях между командами.

5 Парсинг последовательностей команд

5.1 Функция parse_input_with_separators

```
command_sequence_t *parse_input_with_separators(const char *input) {
    if (input == NULL || strlen(input) == 0) {
        return NULL;
    }

    //
    command_t *full_cmd = parse_input(input);
    if (full_cmd == NULL) {
        return NULL;
    }

    command_sequence_t *seq = malloc(sizeof(command_sequence_t));
    if (seq == NULL) {
        perror("malloc");
        free_command(full_cmd);
        return NULL;
    }
    // ...

}
```

Назначение: Парсинг строки с разделителями команд на последовательность.
Алгоритм работы:

1. Парсинг всей строки как единой команды
2. Разделение на подкоманды по разделителям `;`, `&&`, `||`
3. Создание отдельных структур `command_t` для каждой подкоманды
4. Сохранение информации о типах разделителей
5. Копирование перенаправлений в подкоманды

5.2 Структура command_sequence_t

```
typedef struct {
    command_t **commands;     //
    int command_count;        //

    int *separators;          //
                        (0 - ;, 1 - &&, 2 - ||)
} command_sequence_t;
```

6 Обработка перенаправлений и конвейеров

6.1 Функция process_redirections_and_pipes

```
void process_redirections_and_pipes(command_t *cmd, char **words, int *
    word_num) {
    int i = 0;
```

```
    while (i < *word_num) {
        //
        if (words[i] == NULL || strlen(words[i]) == 0 ||
    is_command_separator(words[i])) {
            i++;
            continue;
        }

        int tokens_to_remove = 0;
        char *filename = NULL;

        //
        if (strcmp(words[i], ">") == 0) {
            //
            if (i + 1 >= *word_num) {
                fprintf(stderr, "Error: expected filename after '>'\n");
                i++;
                continue;
            }
            // ...

        }
        // ...
    }
}
```

Назначение: Обработка и извлечение операторов перенаправления из массива слов.
Поддерживаемые операторы:

- `>` - перенаправление вывода (перезапись)
- `>>` - перенаправление вывода (дополнение)
- `<` - перенаправление ввода
- `2>` - перенаправление stderr (перезапись)
- `2>>` - перенаправление stderr (дополнение)
- `2>&1` - объединение stderr с stdout

7 Функции освобождения памяти

7.1 Функция `free_command`

```
void free_command(command_t *cmd) {
    if (cmd == NULL) return;

    //
    for (int i = 0; i < cmd->word_num; i++) {
        free(cmd->words[i]);
    }
    free(cmd->words);

    //
    free(cmd->input_file);
```

```
    free(cmd->output_file);
    free(cmd->error_file);

    //                                          (                  )
    if (cmd->pipeline != NULL) {
        for (int i = 0; i < cmd->pipeline_count; i++) {
            for (int j = 0; cmd->pipeline[i][j] != NULL; j++) {
                free(cmd->pipeline[i][j]);
            }
            free(cmd->pipeline[i]);
        }
        free(cmd->pipeline);
    }

    free(cmd);
}
```

Назначение: Полное освобождение ресурсов структуры команды.

7.2 Функция free_command_sequence

```
void free_command_sequence(command_sequence_t *seq) {
    if (seq == NULL) return;

    for (int i = 0; i < seq->command_count; i++) {
        free_command(seq->commands[i]);
    }
    free(seq->commands);
    free(seq->separators);
    free(seq);
}
```

Назначение: Освобождение ресурсов последовательности команд.

8 Функции отладки

8.1 Функция print_command

```
void print_command(const command_t *cmd) {
    if (cmd == NULL) {
        printf("Command: NULL\n");
        return;
    }

    printf("Command words: ");
    for (int i = 0; i < cmd->word_num; i++) {
        printf("[%s] ", cmd->words[i]);
    }
    printf("\n");

    printf("fonius: %s\n", cmd->fonius ? "yes" : "no");
    printf("Input file: %s\n", cmd->input_file ? cmd->input_file : "NULL");
    printf("Output file: %s\n", cmd->output_file ? cmd->output_file : "NULL"
   );
    printf("Error file: %s\n", cmd->error_file ? cmd->error_file : "NULL");
    printf("Append output: %s\n", cmd->append_output ? "yes" : "no");
    printf("Append error: %s\n", cmd->append_error ? "yes" : "no");
```

```
    printf("Merge output: %s\n", cmd->merge_output ? "yes" : "no");
    printf("Pipeline count: %d\n", cmd->pipeline_count);
}
```

Назначение: Вывод отладочной информации о команде.

8.2 Функция print_command_sequence

```
void print_command_sequence(const command_sequence_t *seq) {
    if (seq == NULL) {
        printf("Command sequence: NULL\n");
        return;
    }

    printf("Command sequence (%d commands):\n", seq->command_count);
    for (int i = 0; i < seq->command_count; i++) {
        printf("  Command %d: ", i + 1);
        for (int j = 0; j < seq->commands[i]->word_num; j++) {
            printf("[%s] ", seq->commands[i]->words[j]);
        }
        printf("\n");

        //

        if (seq->commands[i]->input_file)
            printf("    Input: %s\n", seq->commands[i]->input_file);
        if (seq->commands[i]->output_file)
            printf("    Output: %s (%s)\n", seq->commands[i]->output_file,
                   seq->commands[i]->append_output ? "append" : "trunc");
        // ...
    }
}
```

Назначение: Вывод отладочной информации о последовательности команд.

9 Примеры работы парсера

9.1 Пример 1: Простая команда

```
Ввод: "ls -la"
Результат:
Command words: [ls] [-la]
fonius: no
Input file: NULL
Output file: NULL
Error file: NULL
```

9.2 Пример 2: Команда с перенаправлениями

```
Ввод: "ls -la > output.txt 2>&1"
Результат:
Command words: [ls] [-la]
fonius: no
Input file: NULL
Output file: output.txt
```

```
Error file: NULL
Merge output: yes
```

9.3 Пример 3: Последовательность команд

```
Ввод: "ls && pwd || echo error"
Результат:
Command sequence (3 commands):
  Command 1: [ls]
  Separator: &&
  Command 2: [pwd]
  Separator: ||
  Command 3: [echo] [error]
```

9.4 Пример 4: Команда в фоновом режиме

```
Ввод: "sleep 10 &"
Результат:
Command words: [sleep] [10]
fonius: yes
```

10 Особенности реализации

10.1 Обработка специальных символов

- Символы `@#%!& $ ^ ; : , ( ) { } [ ]` считаются разделительными
- Двойные символы обрабатываются как единые токены
- Проверка корректности расположения операторов

10.2 Управление памятью

- Все строки копируются с помощью `strdup()`
- Освобождение памяти при ошибках парсинга
- Рекурсивное освобождение вложенных структур

10.3 Обработка ошибок

- Проверка границ массивов
- Валидация синтаксиса перенаправлений
- Корректная обработка неожиданного конца строки

11 Взаимодействие с другими модулями

11.1 Зависимости

- `shell.h` - объявления структур и констант
- `executor.c` - использует результаты парсинга для выполнения команд
- `main.c` - передает входные данные для парсинга

11.2 Поток данных

1. `main.c` → Входная строка от пользователя
2. `parse_input()` → Структура `command_t`
3. `executor.c` → Выполнение разобранной команды

12 Заключение

Модуль `parcer.c` реализует сложный и надежный парсер командной строки, который корректно обрабатывает все основные конструкции shell'а. Парсер обеспечивает:

- Полную поддержку стандартного синтаксиса командной строки
- Обработку специальных символов и операторов
- Поддержку перенаправлений ввода/вывода
- Парсинг последовательностей и конвейеров команд
- Надежное управление памятью и обработку ошибок

Модуль успешно интегрируется с другими компонентами системы и обеспечивает надежную основу для выполнения пользовательских команд.