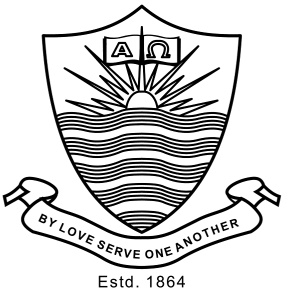


FORMAN CHRISTIAN COLLEGE

(A CHARTERED UNIVERSITY)

Project Title:

Custom Shell Interpreter

Team Members:

- **Hamza Tarar**
- **Ali Sudaif**
- **Joshua Sadaqat**
- **Saud ijaz**

Course Information:

- Course: **Comp 301**
- Section: **A**
- Submitted to: **Sir Salman Chaudhry**

Table of Contents

Project Overview: 3

Project Objectives: 3

Key Components: 3

Command Line Interpreter (CLI) 3

User Interface: 3

Additional Components: 3

Implementation Phases: *4*

Phase-1: 4

First Shell Version 4

Compilation 4

Execution 4

Basic Functionality 4

Exit 4

Phase-2: 5

Simple Commands 5

Source Abstraction 5

Phase-3: 6

1. Environment Variables 6

2. Shell Variables 6

3. Word Expansion 7

Compilation: 7

Testing: 8

Phase-4: 9

Overview: 9

Symbol Table Functions: 9

Symbol Table Stack Functions: 10

Initializing Symbol Tables: 10

Builtin Utilities: 10

Phase-5: 12

Introduction to Word Expansion 12

Word Expansion Process 12

Working with Words 12

Helper Functions 12

Individual Word Expansion Functions 12

Updating the Scanner and Executor 14

Compilation and Usage 14

Screenshots *15*

Conclusion *16*

Project Overview:

The Custom Shell Interpreter project aims to develop a fully functional Linux shell from scratch, providing users and programmers with insights into the intricacies of shell operation. Inspired by the historical evolution of Unix shells, the project focuses on understanding the parsing and execution of commands, as well as implementing essential features like word expansion, history substitution, and I/O redirection.

Project Objectives:

Since the inception of Unix, the shell has been a critical component of the user's interaction with the operating system. Over time, shells evolved to encompass a myriad of features, including I/O redirection, command pipelines, word expansion, history substitution, loops, and conditional expressions. The purpose of this project is to demystify the inner workings of Linux shells by building one from scratch, providing a tangible and comprehensive learning experience.

Key Components:

Command Line Interpreter (CLI)

The core of the Linux shell comprises a Command Line Interpreter, which consists of two fundamental parts:

1. Parser (Front-end): Responsible for reading and tokenizing user commands.
2. Executor (Back-end): Executes parsed commands, completing the command execution cycle.

The parser scans input, breaks it into tokens, and creates an Abstract Syntax Tree (AST) as a high-level representation of the user's command. The executor processes the AST and carries out the parsed command.

User Interface:

The shell operates interactively through a gtk+-3.0. The user interface involves reading input, parsing and executing it, and looping to read the next command until specific exit commands are entered.

Additional Components:

1. Symbol Table: Manages information about variables, including their values and attributes.
2. History Facility: Enables users to access and re-execute recently entered commands.

3. Built-in Utilities: Special commands integrated into the shell program, such as cd, fg, and bg.

Implementation Phases:

Phase-1:

First Shell Version

The initial version of the shell provides a basic gtk loop, echoing user input back to the screen. It establishes a foundation for future enhancements and functionalities.

Compilation

Compile the shell using the following command:

gcc -o shell main.c prompt.c

Execution

Run the shell:

./shell

Basic Functionality

1. Displays PS1 prompt for entering commands.
2. Reads user input, echoing it back.
3. Handles multiline input with PS2 prompt.

Exit

To exit the shell, type:

Exit

Phase-2:

In Part II of the Custom Shell Interpreter project, we enhance our initial shell by introducing the ability to parse and execute simple commands. The focus is on understanding and implementing the necessary components for lexical scanning, tokenization, parsing, and execution of commands.

Simple Commands

A simple command is a list of words separated by whitespace characters. The first word is the command name, and the subsequent words, if present, constitute the command's arguments.To enable command execution, our shell must perform the following steps:

1. Scanning Input:

- Lexical scanning, handled by the lexical scanner or the scanner.

- Retrieving the next character, ungetting a character, peeking at the next character, and skipping whitespace characters are essential scanner functions.

2. Tokenizing Input:

- Tokenization, performed by the tokenizer.

- Creating a structure to represent tokens, containing information about the token's source, length, and text.

Source Abstraction

The `source.h` and `source.c` files introduce a structure (`struct source_s`) to abstract input. This structure includes a buffer for input text, the size of the text, and the current position within the text. Functions such as `next_char`, `unget_char`, `peek_char`, and `skip_white_spaces` aid in managing input.

Tokenization

The `scanner.h` and `scanner.c` files define a `struct token_s` structure to represent tokens. The global variables `tok_buf`, `tok_bufsize`, and `tok_bufindex` assist in token buffer management.

Functions like `add_to_buf`, `create_token`, `free_token`, and `tokenize` collectively handle tokenization. The `tokenize` function reads input characters, identifies token boundaries, and creates tokens for further processing.

Compilation

No compilation is required at this stage, as our shell is not yet ready to parse and execute commands. Compilation will be performed after implementing the parser and executor in Part III.

Phase-3:

In the third part of our shell project, significant enhancements have been made to handle environment variables, shell variables, and word expansion. Below are the key components and functionalities implemented in this part:

1. Environment Variables

File: `environment.c`

The `environment.c` file contains the implementation of functions related to environment variables. Environment variables are system-wide variables that store information about the environment in which the shell is running. The following function has been added:

```char *get_env_var(char *name);```

This function retrieves the value of a specified environment variable by its name using the `getenv` function from the standard library.

2. Shell Variables

File: `shell.h`, `variables.c`

A mechanism for handling shell variables, which are variables specific to the shell, has been introduced. Shell variables can be set and retrieved within the shell. The following structures and functions are added:

```
// In `shell.h`
struct shell_var_s
{
   char *name;
   char *value;
};
void set_shell_var(char *name, char *value);
char *get_shell_var(char *name);
```

- `set_shell_var`: Sets the value of a shell variable identified by its name.
- `get_shell_var`: Retrieves the value of a shell variable given its name.
```

File: `variables.c`

The `variables.c` file contains the implementation of these functions. Shell variables are maintained in a dynamic structure to allow for easy addition and retrieval.

3. Word Expansion

****File: `expansion.c`****

The `expansion.c` file introduces the concept of word expansion. Word expansion is a process that occurs before executing a command, where certain patterns or variables in a word are replaced with their corresponding values. The following function is added:

```c
struct node_s *expand_word(struct node_s *word);
```

This function takes a word node from the abstract syntax tree (AST) and performs necessary expansions, such as replacing environment variables and shell variables with their values.

How to Use:

- ****Environment Variables:****

  -Retrieve the value of an environment variable using `get_env_var("VAR_NAME")`.

- ****Shell Variables:****

  - Set a shell variable: `set_shell_var("VAR_NAME", "value")`.

  - Retrieve the value of a shell variable: `get_shell_var("VAR_NAME")`.

- ****Word Expansion:****

  - The `expand_word` function is automatically invoked during the parsing and execution of commands. It handles expansions within words, providing a seamless experience for users.

Compilation:

Compile the shell by running the following command:

```bash
```

gcc -o shell main.c source.c scanner.c parser.c node.c executor.c prompt.c environment.c variables.c expansion.c

```

## Testing:

**After compilation, run the shell:**

```**bash**

./shell

```

Test various commands, including those involving environment and shell variables, to ensure proper functionality.
```

Phase-4:

In Part IV of our Linux shell project, we introduce symbol tables to enhance the management of shell variables and functions. Symbol tables are essential data structures used by compilers and interpreters to store variables and their values. Each entry in the symbol table consists of a key (variable name) and an associated value (variable value). Symbol tables enable efficient variable retrieval, type checking, scoping rules enforcement, and variable exportation to external commands.

Overview:

- **Symbol Table Implementation:**

- The symbol table is implemented using linked lists, chosen for simplicity and efficiency.

- Three main structures are introduced: `enum symbol_type_e`, `struct symtab_entry_s`, and `struct symtab_s`.

- Symbol table entries can represent shell variables (`SYM_STR`) or shell functions (`SYM_FUNC`).

- The symbol table structure maintains a linked list of entries.

Symbol Table Functions:

1. Initialization:

- `init_symtab()`: Initializes the symbol table stack, creating the global symbol table.

2. Manipulating Symbol Tables:

- `new_symtab(int level)`: Creates a new symbol table with the specified level.

- `free_symtab(struct symtab_s *symtab)`: Frees the memory used by a symbol table.

- `dump_local_symtab()`: Prints the contents of the local symbol table for debugging purposes.

3. Manipulating Symbol Table Entries:

- `add_to_symtab(char *symbol)`: Adds a new entry to the local symbol table.

- `rem_from_symtab(struct symtab_entry_s *entry, struct symtab_s *symtab)`: Removes an entry from the symbol table.

- `do_lookup(char *str, struct symtab_s *symtable)`: Performs a lookup for a variable with the given name.

- `get_symtab_entry(char *str)`: Searches the entire symbol table stack for an entry by name.

- `symtab_entry_setval(struct symtab_entry_s *entry, char *val)`: Sets the value of a symbol table entry.

Symbol Table Stack Functions:

Manipulating Symbol Table Stack:

- `symtab_stack_add(struct symtab_s *symtab)`: Adds a symbol table to the stack.

- `symtab_stack_push()`: Pushes a new symbol table onto the stack.

- `symtab_stack_pop()`: Pops the top symbol table from the stack.

2. Accessing Symbol Tables:

- `get_local_symtab()`: Returns a pointer to the local symbol table.

- `get_global_symtab()`: Returns a pointer to the global symbol table.

- `get_symtab_stack()`: Returns a pointer to the symbol table stack.

Initializing Symbol Tables:

- The `initsh()` function in `initsh.c` initializes the symbol table stack, creates the global symbol table, and populates it with environment variables.

- PS1 and PS2 variables are added to store the prompt strings.

Builtin Utilities:

- A `struct builtin_s` structure is introduced to store information about builtin utilities.

- The `dump` utility is added to print the contents of the local symbol table.

Executor Update:

- The executor now checks if a command is a builtin utility and calls the corresponding function if found.

- This ensures that our new `dump` utility is executed internally.

Compilation:

Compile the shell by running:

```bash
gcc -o shell executor.c initsh.c main.c node.c parser.c prompt.c scanner.c source.c builtins/builtins.c builtins/dump.c symtab/symtab.c
```

Testing:

After compilation, run the shell:

```bash
./shell
```

Test various commands, including the `dump` utility, to observe symbol table changes.

Phase-5:

Introduction to Word Expansion

In Part V of the project, we delve into the intricate process of word expansion within a Linux shell. Word expansion involves interpreting special characters within a command line, performing various expansions, and transforming the command line into an executable form.

Word Expansion Process

The word expansion process consists of several steps, each handling different types of expansions. These include tilde expansion, parameter expansion, arithmetic expansion, command substitution, field splitting, pathname expansion, and quote removal.

Working with Words

To manage the expanded words, a special structure called `struct word_s` is introduced. This structure contains fields such as `data` (string representing the word), `len` (length of the data field), and `next` (pointer to the next word or NULL if it's the last word).

Two functions, `make_word` and `free_all_words`, assist in allocating and freeing memory for these word structures.

Helper Functions

Before delving into specific word expansions, various helper functions are defined to aid in the complexity of the process. These functions include:

- `wordlist_to_str`: Converts a linked list of expanded words to a single string.
- `delete_char_at`: Removes a character at the given index from a string.
- `is_name`: Checks if a string represents a valid variable name.
- `find_closing_quote`: Finds the index of the closing quote character in a word.
- `find_closing_brace`: Finds the index of the closing brace character.
- `substitute_str`: Substitutes a substring with another string.
- `substitute_word`: Calls other word expansion functions.

Additional string-handling functions are defined in the `strings.c` source file.

Individual Word Expansion Functions

The following functions handle specific word expansions:

Tilde Expansion (`tilde_expand`)

- Replaces a tilde character with the pathname of the user's home directory.
- Supports tilde prefixes, including optional usernames.

Parameter Expansion (`var_expand`)

- Replaces the name of a shell variable with its value.
- Utilizes parameter expansion modifiers defined by POSIX.
- Handles cases where variable names are surrounded by curly braces.

Command Substitution (`command_substitute`)

- Executes a command and replaces the command substitution with its output.
- Supports both `$()` and `` styles of command substitution.

Arithmetic Expansion (`arithm_expand`)

- Evaluates arithmetic expressions enclosed in `$((...))`.
- Supports signed long integer arithmetic (floating-point and functions not supported).
- Converts expressions to Reverse Polish Notation (RPN) for parsing.

Field Splitting (`field_split`)

- Splits the results of parameter, command, and arithmetic expansions into fields.
- Utilizes the value of the `$IFS` shell variable to determine field boundaries.

Pathname Expansion (`pathnames_expand`)

- Matches file names with globbing patterns containing `*`, `?`, and `[]`.
- Excludes special names `.` and `..` from the matched file names.

Quote Removal (`remove_quotes`)

- Removes quotes (backslash, single, double) from expanded words.
- Preserves the literal meaning of characters within quoted strings.

Main Word Expansion Function (`word_expand`)

The `word_expand` function ties all these word expansion functionalities together. It scans the input word, identifies special characters, and calls the appropriate expansion functions. The sequence includes tilde expansion, parameter expansion, command substitution, arithmetic expansion, field splitting, pathname expansion, and quote removal.

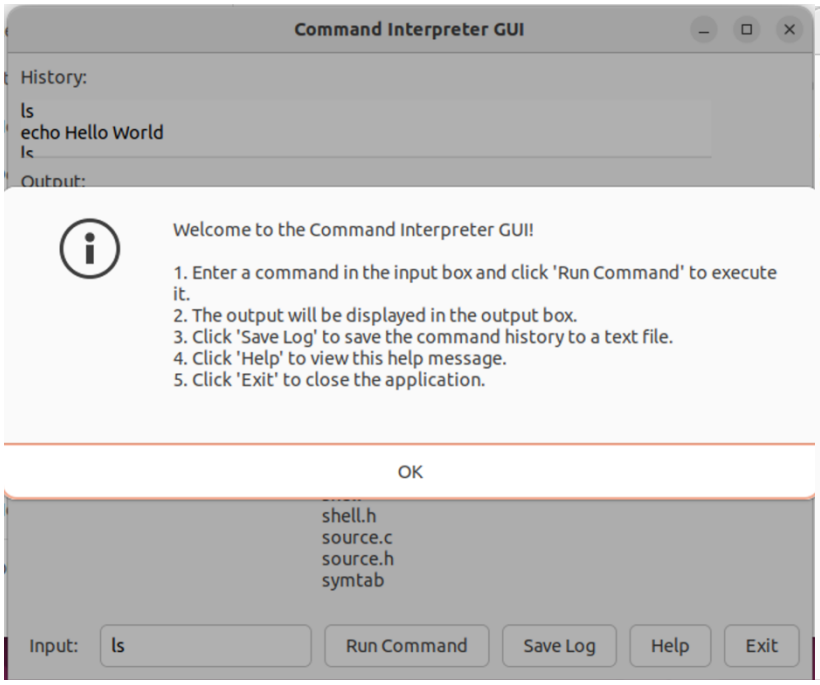
Updating the Scanner and Executor

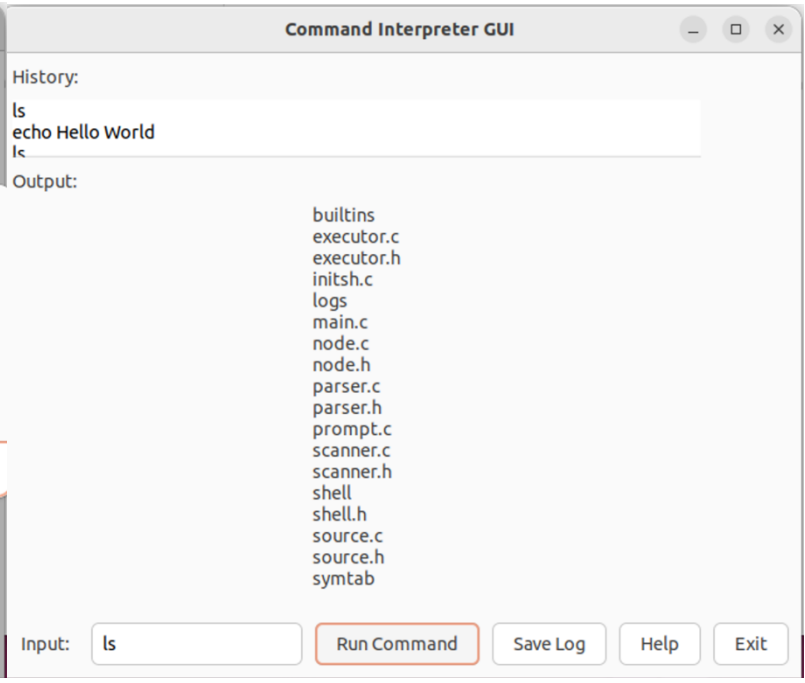
To integrate word expansion into the shell's functionality, updates are made to the scanner and executor components. The scanner is enhanced to recognize and skip over quoted strings and escaped characters. The executor is modified to perform word expansion on command arguments and support more than 255 arguments.

Compilation and Usage

The shell is compiled using the provided makefile, resulting in an executable named `shell`. Users can interact with the shell and observe the effects of various word expansions on command line inputs.

Screenshots





Conclusion

Part V completes the implementation of word expansion in the Linux shell project, enhancing its functionality and enabling a more sophisticated interpretation of command lines. Users are encouraged to experiment with the shell, applying different word expansions and comparing results with default shell behavior.