CUSTOM LINUX SHELL

Technical Report

Haoting Chen

[ANU] | [U7227871]

1. Introduction

The program "anubis.c" is a basic command line interpreter (CLI) or shell that can run programs given an input command containing program path(s) or name. The inputs can be either read from the standard input (interactive mode) or a file (batch mode). In addition to running a single program and output to the screen, "anubis.c" can also run multiple programs in parallel, chain them into a pipeline, or redirect outputs. "anubis.c" also supports some common built-in commands, including "cd", "path", and "exit" to help users navigate through different directories to select a program. A detailed specification can be found in "./anubis/README.md".

2. Custom Data Types

To facilitate programming, two custom data types (struct), "string_list" and "pid_list" and their associated helper functions have been used through the code. (line 38 to line 142).

string_list is the extended version of an array of strings that can automatically allocate memory when created and keep tracking how much memory is in use so that it can allocate more when needed. It is mainly used for holding tokens. The following are its utility functions.

pid_list is similar to "string_list", but it is used for a parent process to manage child processes, especially when running parallel programs.

string_list_create()	Create a list by allocating some memory
string_list_free()	Free all memory for the list
string_list_add()	Add an element to the list and allocate more memory if needed
string_list_sublist()	Get a sub-list (deep copy) given the indexes
string_list_print()	Print the list

Throughout the report, a list is represented by { }, while an array is represented by [ ].

3. Overall Structure

The flowchart shown in Figure 1 gives a simplistic view of the program. In “main()” (line 449 to line 508), the program first determines whether the input is from the user keyboard (standard input) or a file. Then, go into a loop to keep reading a line of command line until the end of the file is read, or the user types an “exit”.

In each loop, the command (string) is tokenized into a list of strings, “run_cmd_built_in()” tries running the tokenized command as a built-in command. If it turns out that the command does not match any of the built-in commands, the program continues to send the commands into a parser, where the command is split into several sub-commands by the operator "|", ">", and "&" and the sub-commands are executed according to the precedence of the operator binding them. Suppose that all programs the shell asked to run will eventually terminate, the parser will always reap all running processes and return to main() for another loop.

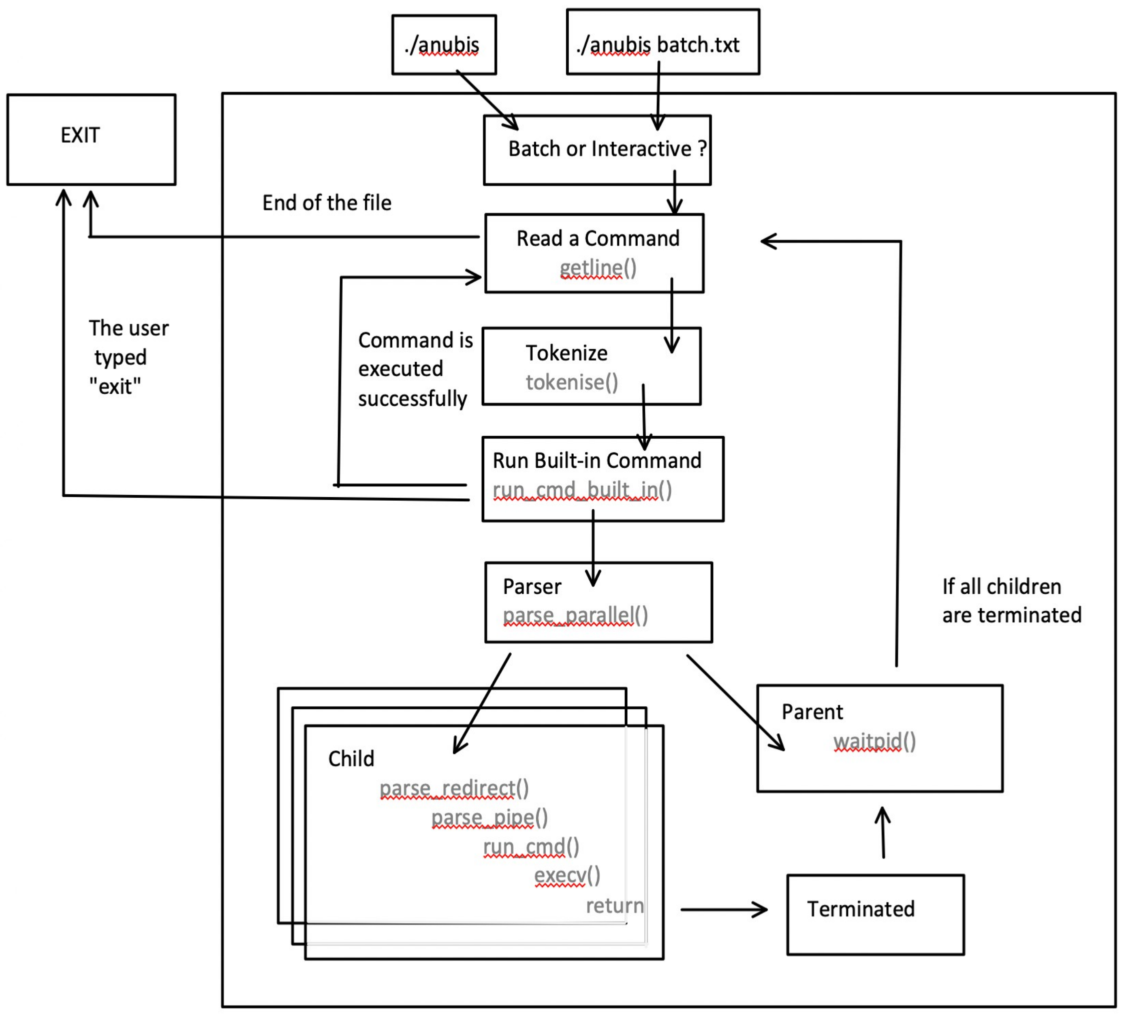


Figure 1: flowchart of main()

3. Tokenizer

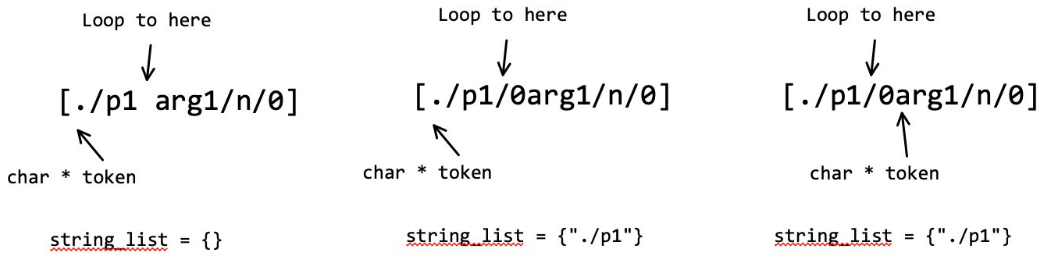
The purpose of the tokenizer (line 149 to line 183) is to turn a string into a list of strings separated by delimiters. Considering the case that the input can be unformatted, e.g.

```
./p1 arg1&./p1>output.txt
```

the tokenizer cannot simply chop the string by space. Instead, the tokenizer loops through each character and makes decisions based on what it is. During this, it keeps a pointer called "char * token” which is the start address of the next available token. It is initially equal to the address of the original string “char * line. In each loop, if:

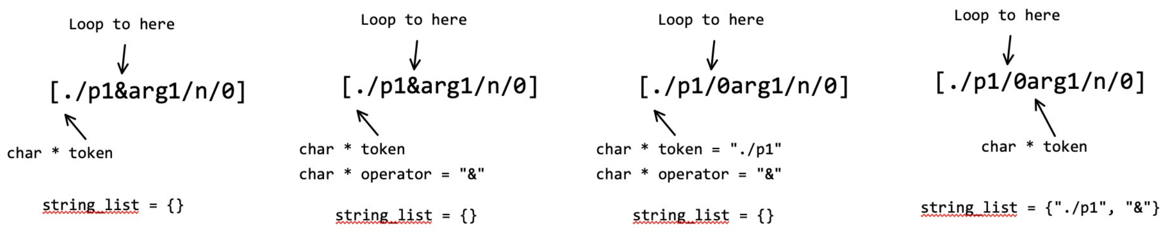
3.1 Character belongs to space, tap, or return.

Rewrite this character with a ‘\0’ so that “char * token” can become a valid substring. Copy and add “token” (if non-empty) into the token list and redirect the pointer “token” to the character after ‘\0’, shown in the figure below.



3.2 Character belongs to &, >, or |

Remember what operator the character belongs to. Rewrite this character with a '\0' so that "token" can become a valid substring and add it to the token list (if non-empty). After that, add the operator token as well. Update the pointer “token” to the character after ‘\0’.

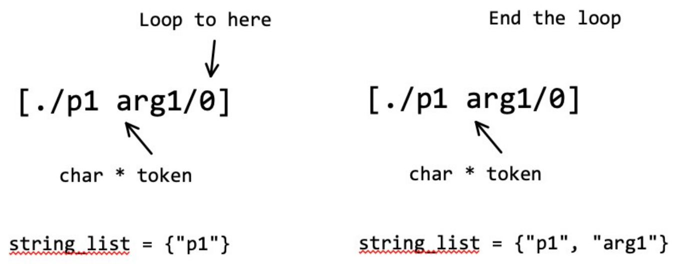


3.3 Other Characters

Continue to the next character without updating either the next-token pointer or the token list.

3.4 Character is ‘\0’

Add the substring "token" to the list if it is not empty.



4. Parser

The parser is the key component of the program that handles all parallel processes, process pipelines, and redirect output. To better explain, an example token list is used.

```
{“p1”, “|”, “p2”, “|”, “p3”, “arg1”, “>”, “output”, “&”, “p3”, “&”,
“p4”, “|”, “p5”}
```

Figure 2 shows the flowchart of the parser, which consists of three parser functions to parse "&", ">", and "|", respectively, in the order of a tree.

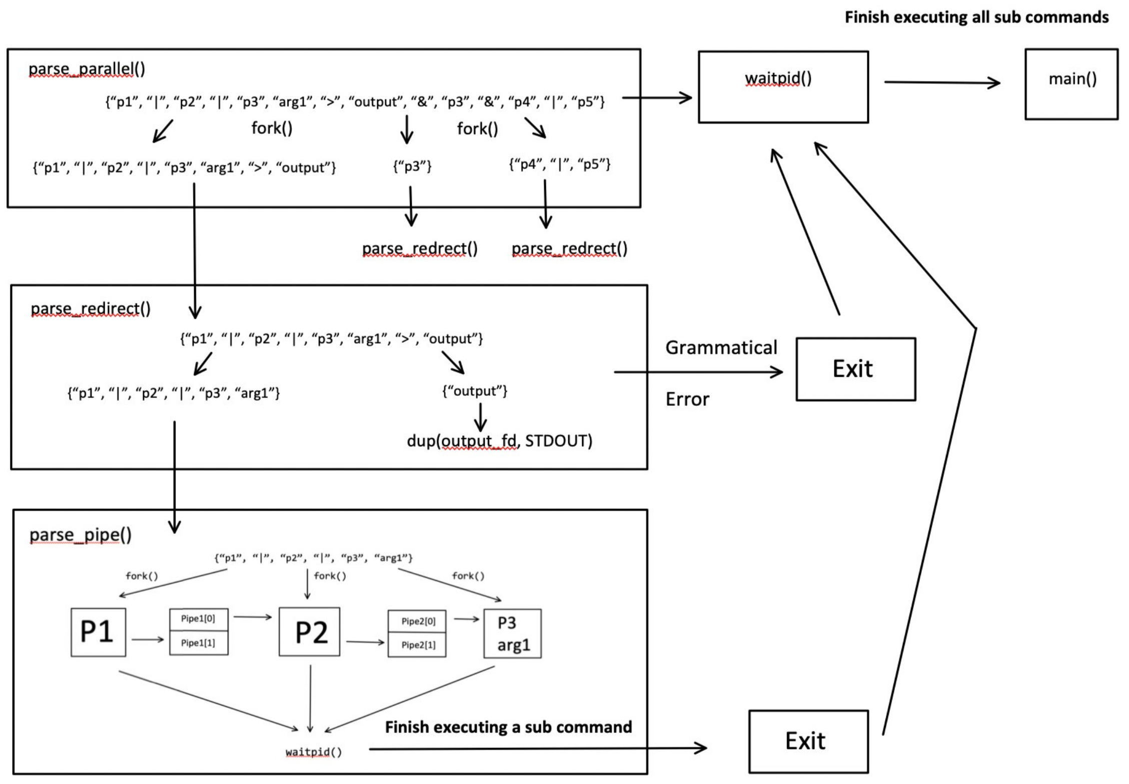

Finish executing all sub commands
parse_parallel()
{"p1", "|", "p2", "|", "p3", "arg1", ">", "output", "&", "p3", "&", "p4", "|", "p5"}
fork()
fork()
{"p1", "|", "p2", "|", "p3", "arg1", ">", "output"}
{"p3"}
{"p4", "|", "p5"}
waitpid()
main()
parse_redrect()
parse_redrect()
parse_redirect()
{"p1", "|", "p2", "|", "p3", "arg1", ">", "output"}
{"p1", "|", "p2", "|", "p3", "arg1"}
{"output"}
dup(output_fd, STDOUT)
Grammatical
Error
Exit
parse_pipe()
{"p1", "|", "p2", "|", "p3", "arg1"}
fork()
fork()
fork()
P1
Pipe1[0]
Pipe1[1]
P2
Pipe2[0]
Pipe2[1]
P3
arg1
waitpid()
Finish executing a sub command
Exit

Figure 2: flowchart of the parser

4.1 parse_parallel()

The first step for parse_parallel() is to split a command (a list of strings) into one or multiple sub-commands by the delimiter "&". In the example, one of the sub-commands is,

```
{"p1", "|", "p2", "|", "p3", "arg1", ">", "output"}
```

The sub-command may contain information about pipeline and output redirection that requires further analysis by the later functions.

After obtaining the sub-commands, parse_parallel() forks a child process for each sub-command, and now all sub-commands will be further parsed and executed in parallel.

In each child process, a sub-command is passed into the next-layer parser function parse_redirect().

Assume that the command (either valid or invalid) that each process carries does not contain an infinite program, all child process should eventually terminate, either finish executing the program(s) specified in the command or be forced to exit because an error occurs during the further parsing of the command.

After all child processes are terminated, parse_parallel() then go back to main().

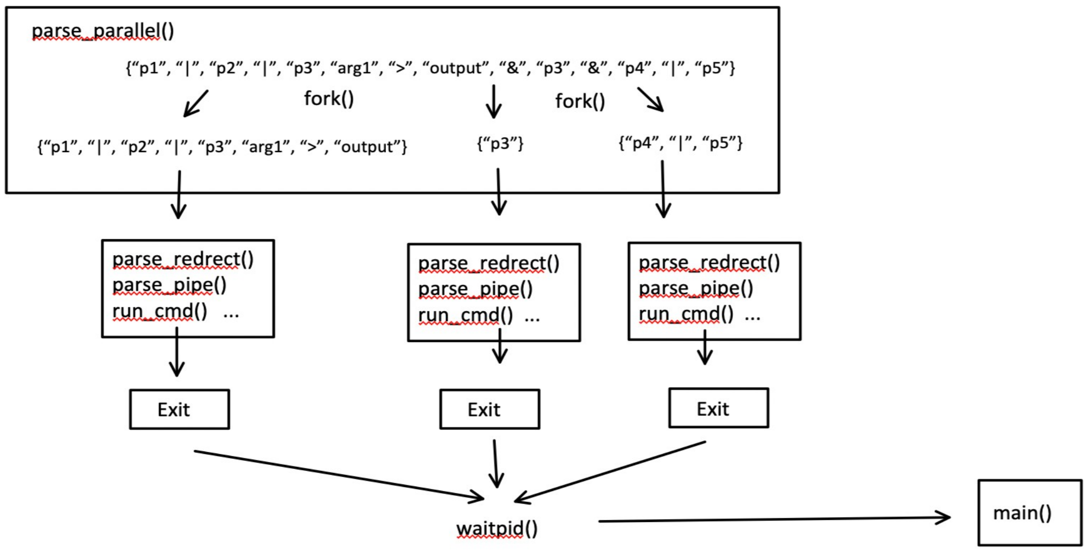


Figure 3: parse_parallel()

4.2 parse_redirect()

When parse_redirect() obtains the input from parse_parallel(), it further splits the command by ">", as shown below.

```
{“p1”, “|”, “p2”, “|”, “p3”, “arg1”, “>”, “output”}
```

Become

```
[{“p1”, “|”, “p2”, “|”, “p3”, “arg1”}, {“output”}]
```

para_redirect() also comes with some grammar checks. For example, the array size must be one (no redirect) or two (with redirect). Within the array, the first element (command to run programs) cannot be empty. The second element (output file) should have string size = 1;

If the user specified a file to redirect, the function redirects the process's standard output to the file using dup().

Then, the function discards the second element (file path) and only passes the first element (command) of the array to the sub-function parse_pipe().

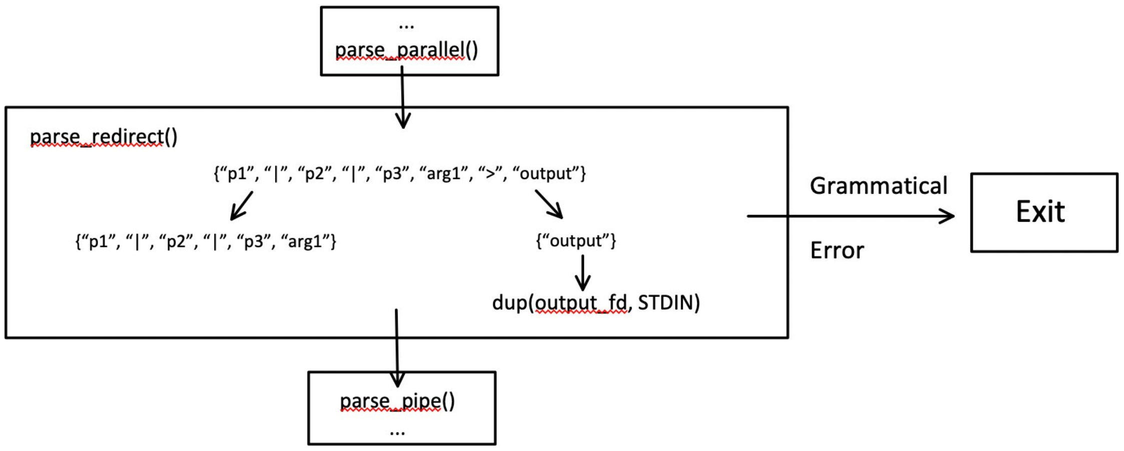


Figure 4: parse_redirect()

4.3 parse_pipe()

The job of parse_pipe() is to chain programs together so that the output of one program goes to the input of the other in order. parse_pipe() read the input from parse_redirect() and split it again by "|", as shown.

```
{“p1”, “|”, “p2”, “|”, “p3”, “arg1”}
```

become

```
[{“p1”}, {“p2”}, {“p3”, “arg1”}]
```

```
pipe_array = [Pipe1[0], Pipe1[1], Pipe2[0], Pipe2[1]]
```

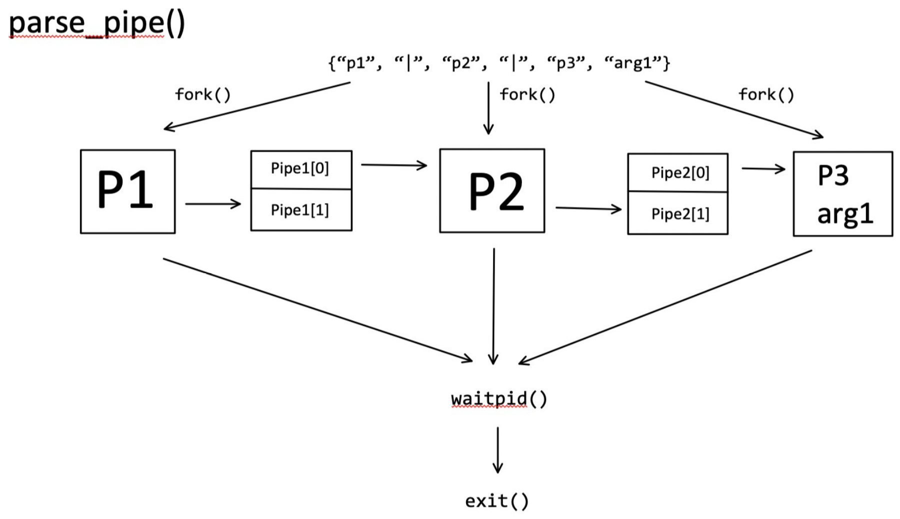


Figure 5: parse_pipe()

A certain number of pipes is created based on the (number of programs – 1). The input and output descriptors of all pipes are stored in an array called “pipe_array”. After opening all pipes, the parent then forks a number of children to run each sub-command.

In each child process, the input and/or output are redirected based on the location program in the pipeline. For example,

1. **First program**: redirect output only
2. **Middle programs**: redirect both output and input
3. **Last program**: redirect input only.

After the redirection, all file descriptors in “pipe_array” should be closed because they are no longer being used, as shown in Figure 6. Then, the child processes call call run_cmd().

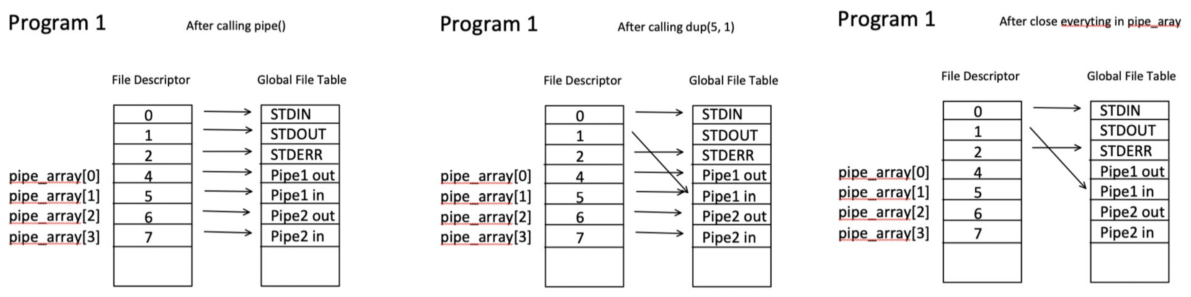


Figure 6: changes of file table in parse_pipe()

The parent process will wait for all child processes to terminate before terminating itself and reaped by the process waiting in the “parse_parallel()” function.

4.4 run_cmd()

run_cmd() is the final layer of the parser. In common shells, some handy programs such as “ls” should run without specifying their address. "anubis" saves a list of directories containing these programs. When run_cmd() gets an input command, it first searches if the program exists in these directories. If so, combine the directories with the program name and call execv(). If not, run it directly using execv() and report an error and exit if execv() returns.

4.5 Design Strategy for Parser

The grammar of the input command determines how to structure the parser. Since there is no bracket in the expression and there are only 3 separators, “|”, “>”, and “&” with distinct precedence, the parser only needs to parse every command exactly 3 times (parse_parallel(), parse_redirct(), and finally parse_pipe()) to reach the most basic expression, i.e. a command to run a single program. In other words, the expression tree obtained by the parser always has depth = 3, as shown in Figure 7. This greatly simplifies the design of the parser.

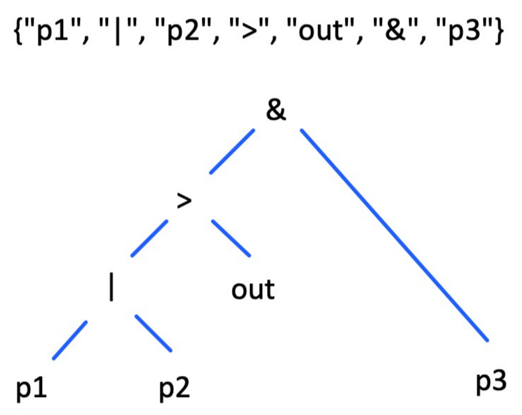


Figure 7: expression tree example

However, if brackets are introduced. parse_parallel(), parse_redirct(), and parse_pipe() may call each other any number of times, and the tree depth = 3 is no longer guaranteed, as shown in Figure 8.

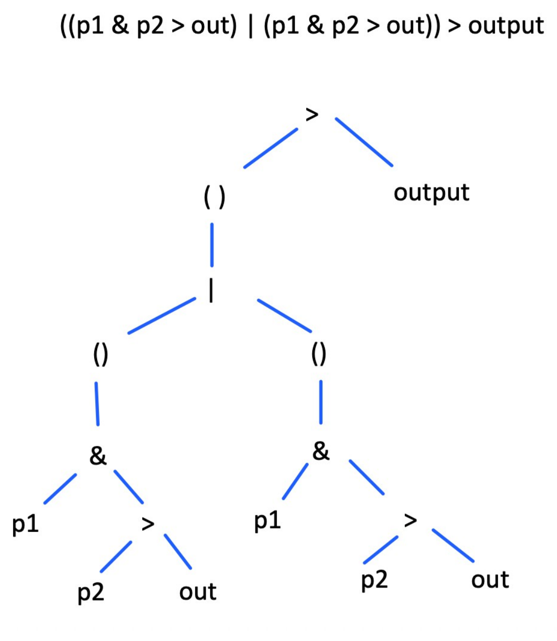


Figure 8: expression tree example (with brackets)

5. Future Work

One future direction for this project is to allow parentheses in input commands, enabling a more flexible input grammar from the current Finite Language Grammar to a Context-Free Grammar. However, to achieve it, the current hard-coded precedence-based linear parser must be rewritten completely into a more generic recursive parser, such as an LL(k) parser. This can pose significant challenges for implementation, especially given the lack of high-level programming language features in C, such as lists, trees, and dynamic memory management. Since a simple finite linear grammar should already be capable of describing most of the commands used by daily users, it may not be a necessary feature to introduce, considering its implementation cost.