Report of Comp3300 Ass

Overview

In this assignment, I implemented the Anubis programmes which is an interactive command line program that supports various features that are in a regular Linux terminal/bash.

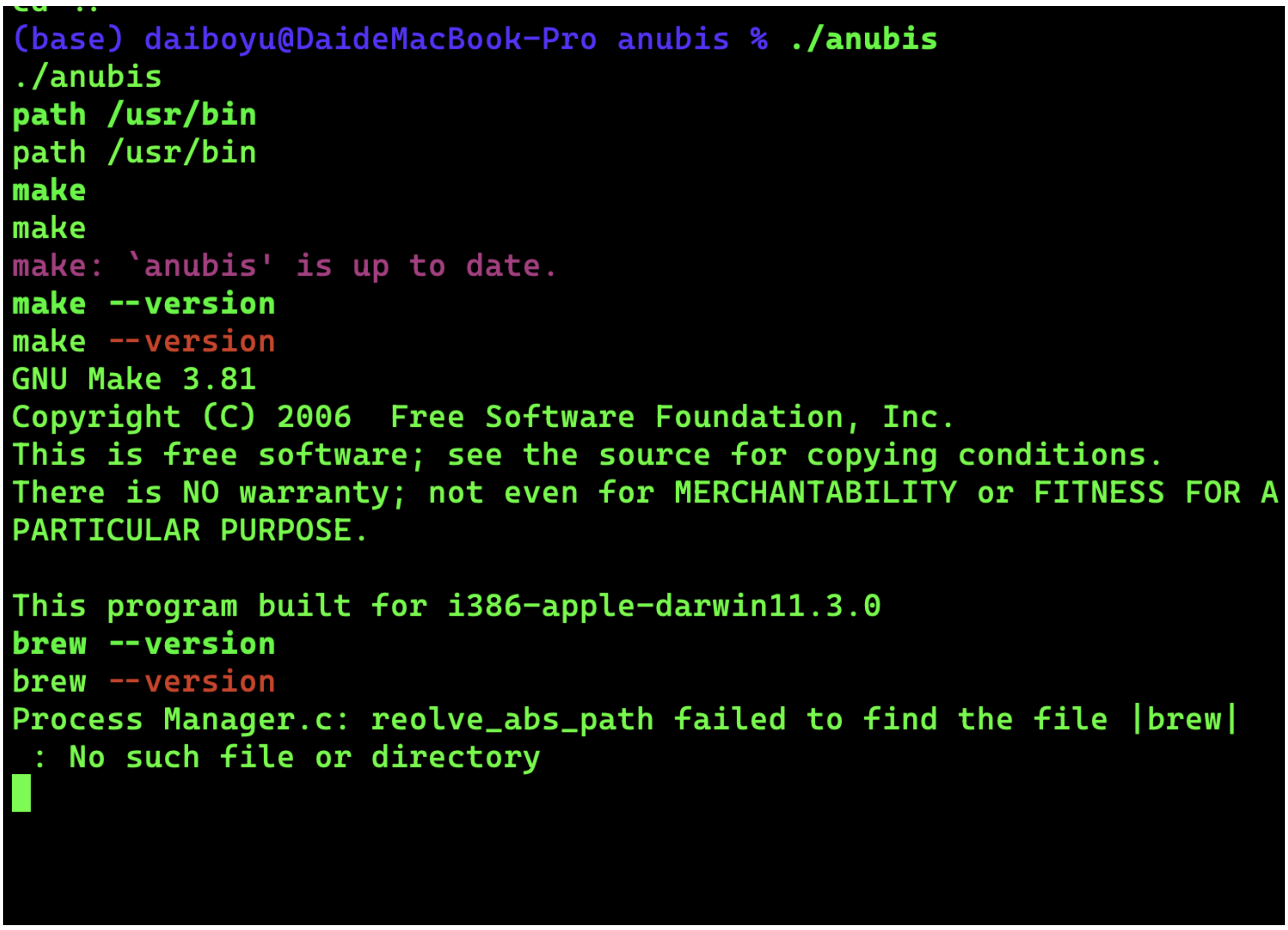
This report will briefly mention some details about the programme's implementation.

Featurese

built-in command: path

Usage: path <serachfolder> | ...

The path command takes multiple input and sets the search path(where anubis look for executable fields) to the input.

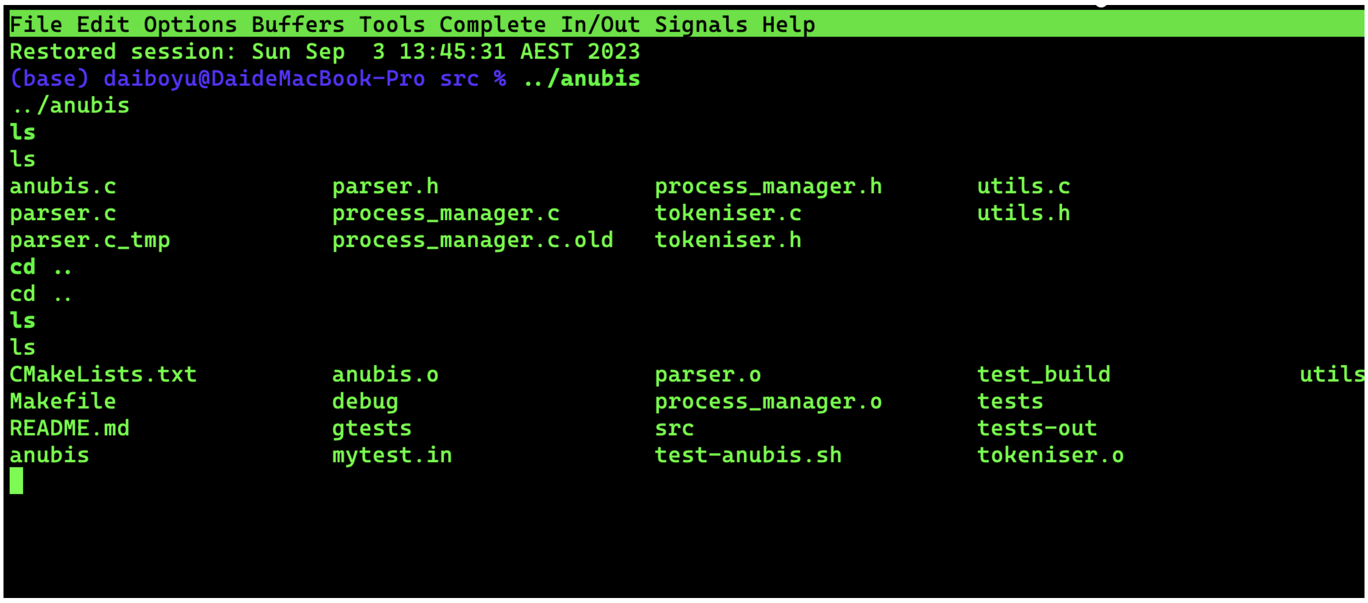


```
(base) daiboyu@DaideMacBook-Pro anubis % ./anubis
./anubis
path /usr/bin
path /usr/bin
make
make
make: `anubis' is up to date.
make --version
make --version
GNU Make 3.81
Copyright (C) 2006  Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

This program built for i386-apple-darwin11.3.0
brew --version
brew --version
Process Manager.c: reolve_abs_path failed to find the file |brew|
 : No such file or directory
```

built-in command: cd

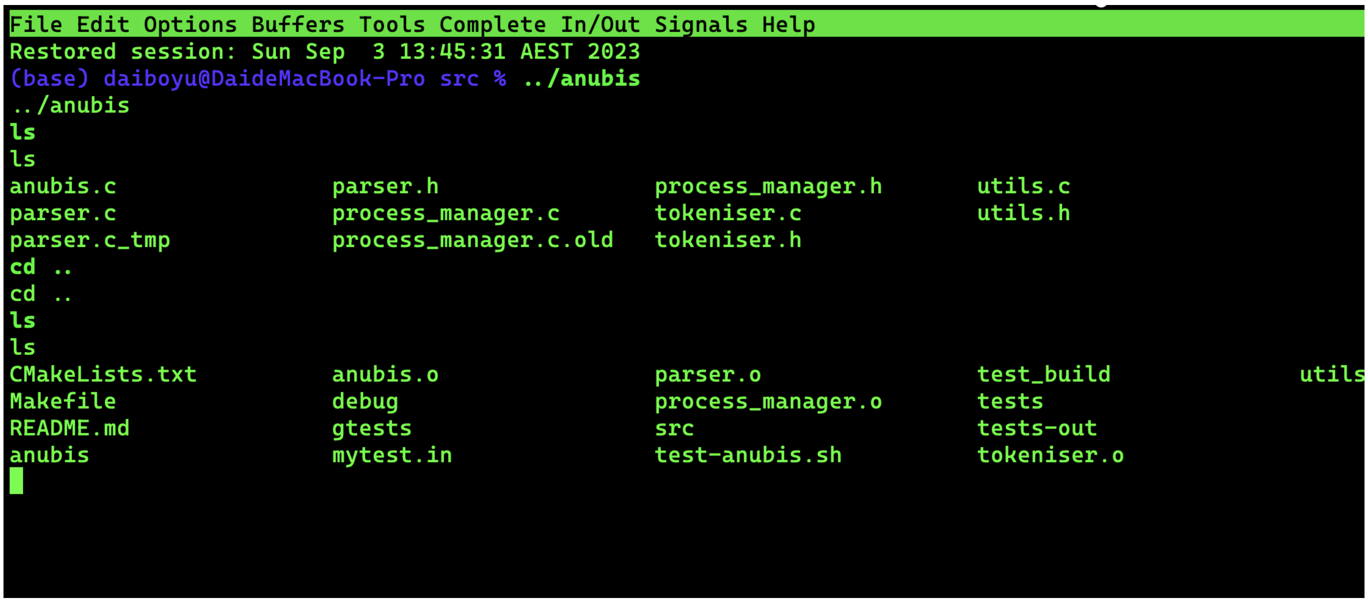
Usage: cd <directory>

Change the working directory of the programme.


```
File Edit Options Buffers Tools Complete In/Out Signals Help
Restored session: Sun Sep  3 13:45:31 AEST 2023
(base) daiboyu@DaideMacBook-Pro src % ../anubis
../anubis
ls
ls
anubis.c                parser.h                process_manager.h       utils.c
parser.c                process_manager.c       tokeniser.c             utils.h
parser.c_tmp            process_manager.c.old   tokeniser.h
cd ..
cd ..
ls
ls
CMakeLists.txt          anubis.o                parser.o                test_build              utils
Makefile                debug                   process_manager.o       tests
README.md               gtests                  src                     tests-out
anubis                  mytest.in               test-anubis.sh          tokeniser.o
```

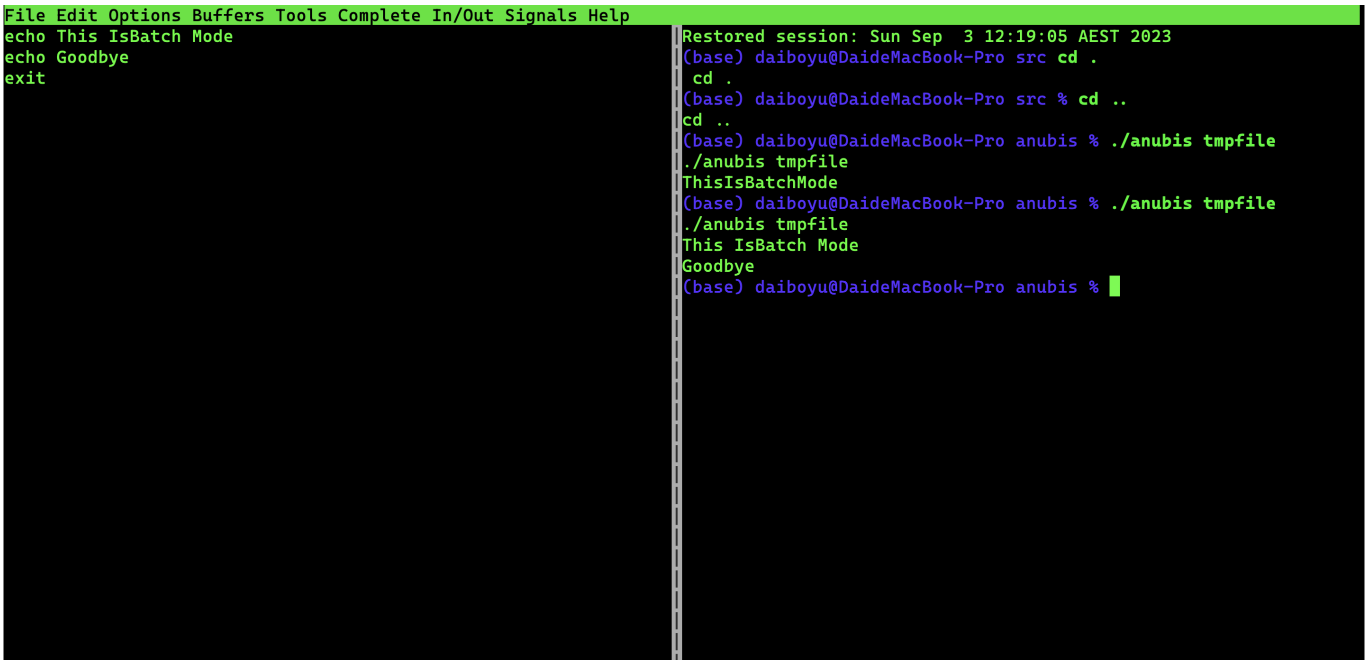
built-in command: exit

Usage: exit the anubis programme


```
File Edit Options Buffers Tools Complete In/Out Signals Help
Restored session: Sun Sep  3 13:45:31 AEST 2023
(base) daiboyu@DaideMacBook-Pro src % ../anubis
../anubis
ls
ls
anubis.c                parser.h                process_manager.h       utils.c
parser.c                process_manager.c       tokeniser.c             utils.h
parser.c_tmp            process_manager.c.old   tokeniser.h
cd ..
cd ..
ls
ls
CMakeLists.txt          anubis.o                parser.o                test_build              utils
Makefile                debug                   process_manager.o       tests
README.md               gtests                  src                     tests-out
anubis                  mytest.in               test-anubis.sh          tokeniser.o
```

Batch mode

The program can accept a file containing lines of instruction and execute the commands line by line inside the file



```
File Edit Options Buffers Tools Complete In/Out Signals Help
echo This IsBatch Mode
echo Goodbye
exit
```

```
Restored session: Sun Sep  3 12:19:05 AEST 2023
(base) daiboyu@DaideMacBook-Pro src cd .
 cd .
(base) daiboyu@DaideMacBook-Pro src % cd ..
cd ..
(base) daiboyu@DaideMacBook-Pro anubis % ./anubis tmpfile
./anubis tmpfile
ThisIsBatchMode
(base) daiboyu@DaideMacBook-Pro anubis % ./anubis tmpfile
./anubis tmpfile
This IsBatch Mode
Goodbye
(base) daiboyu@DaideMacBook-Pro anubis %
```

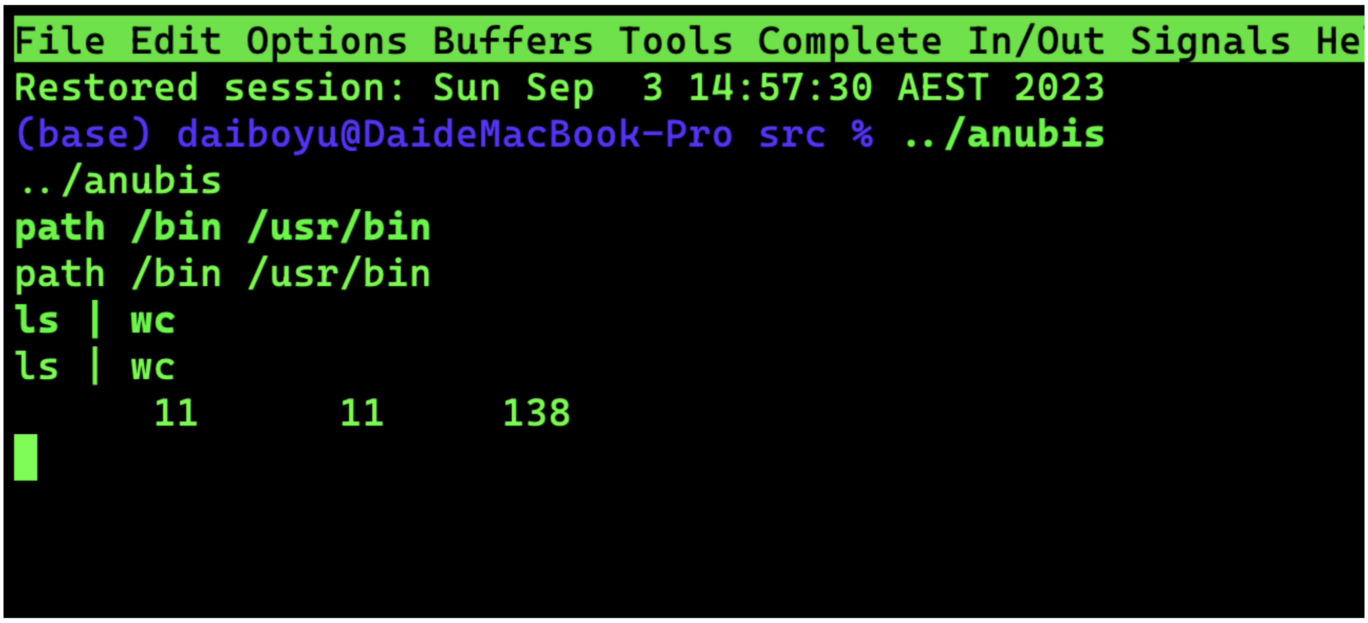
Parallelism

The anubis offers options to the user to run the commands in parallel using the special symbol "&".

cmd1 args1 & cmd2 args2 will allow cmd1 and cmd2 to be executed in parallel(there is no guranteen that cmd2 is going to be executed after cmd1 but due to operation in praparing starting of processes cmd2 is likey to start after cmd1)

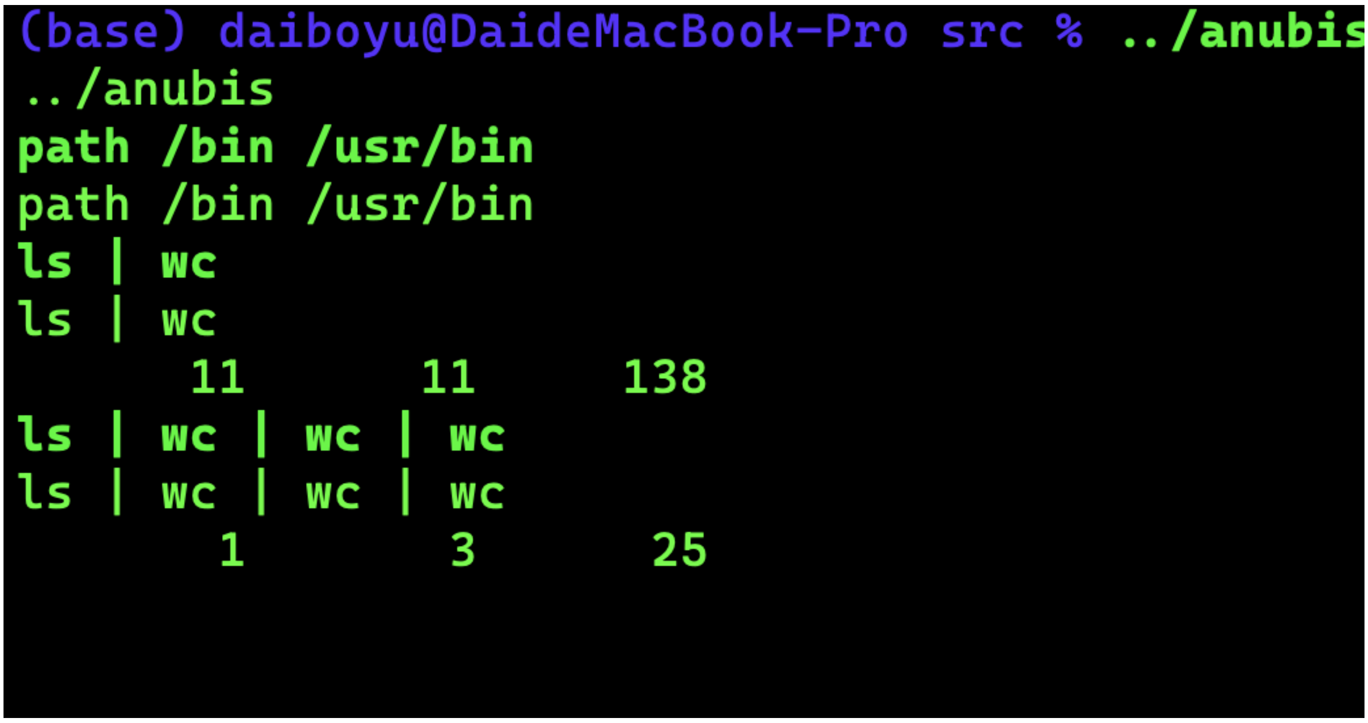
Pipes

The anubis programme allows the user to redirect the stdout of a program to the stdin of another program using **"|"**. For example, **"ls | wc"** would show the word count of the ls command output.



```
File Edit Options Buffers Tools Complete In/Out Signals He
Restored session: Sun Sep  3 14:57:30 AEST 2023
(base) daiboyu@DaideMacBook-Pro src % ../anubis
../anubis
path /bin /usr/bin
path /bin /usr/bin
ls | wc
ls | wc
      11      11     138
```

The number of programs that the usr could chain up does not have a limit. Users could chain up multiple programs using pipe symbols.

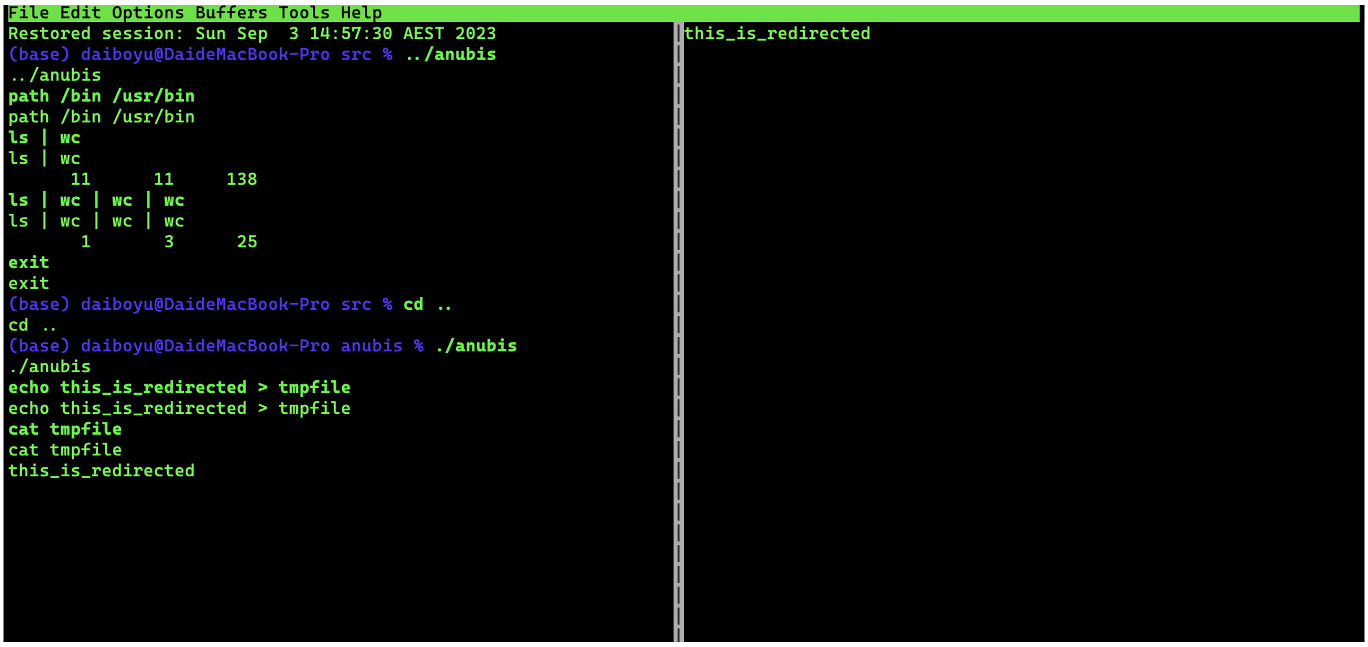


```
(base) daiboyu@DaideMacBook-Pro src % ../anubis
../anubis
path /bin /usr/bin
path /bin /usr/bin
ls | wc
ls | wc
      11      11     138
ls | wc | wc | wc
ls | wc | wc | wc
       1       3      25
```

Output Redirection

Usage: [programme] > [target file]

The anubis program allows the user to redirect the stdout of programs to a file in the file system. If the target file is not found, anubis will create the corresponding file.



```
Restored session: Sun Sep  3 14:57:30 AEST 2023
(base) daiboyu@DaideMacBook-Pro src % ../anubis
../anubis
path /bin /usr/bin
path /bin /usr/bin
ls | wc
ls | wc
      11      11     138
ls | wc | wc | wc
ls | wc | wc | wc
       1       3      25
exit
exit
(base) daiboyu@DaideMacBook-Pro src % cd ..
cd ..
(base) daiboyu@DaideMacBook-Pro anubis % ./anubis
./anubis
echo this_is_redirected > tmpfile
echo this_is_redirected > tmpfile
cat tmpfile
cat tmpfile
this_is_redirected
```

```
this_is_redirected
```

Precedence of execution

A line of command in anubis would follow the following grammar of precedence.

command_exp := name_of_bin and arguments

redir_exp := pipe_exp | pipe_exp > file_name

pipe_exp := command_exp / command_exp | pipe_exp parallel_exp := redir_exp | redir_exp & parallel_exp

Implementation details & documentation

Anubis.c

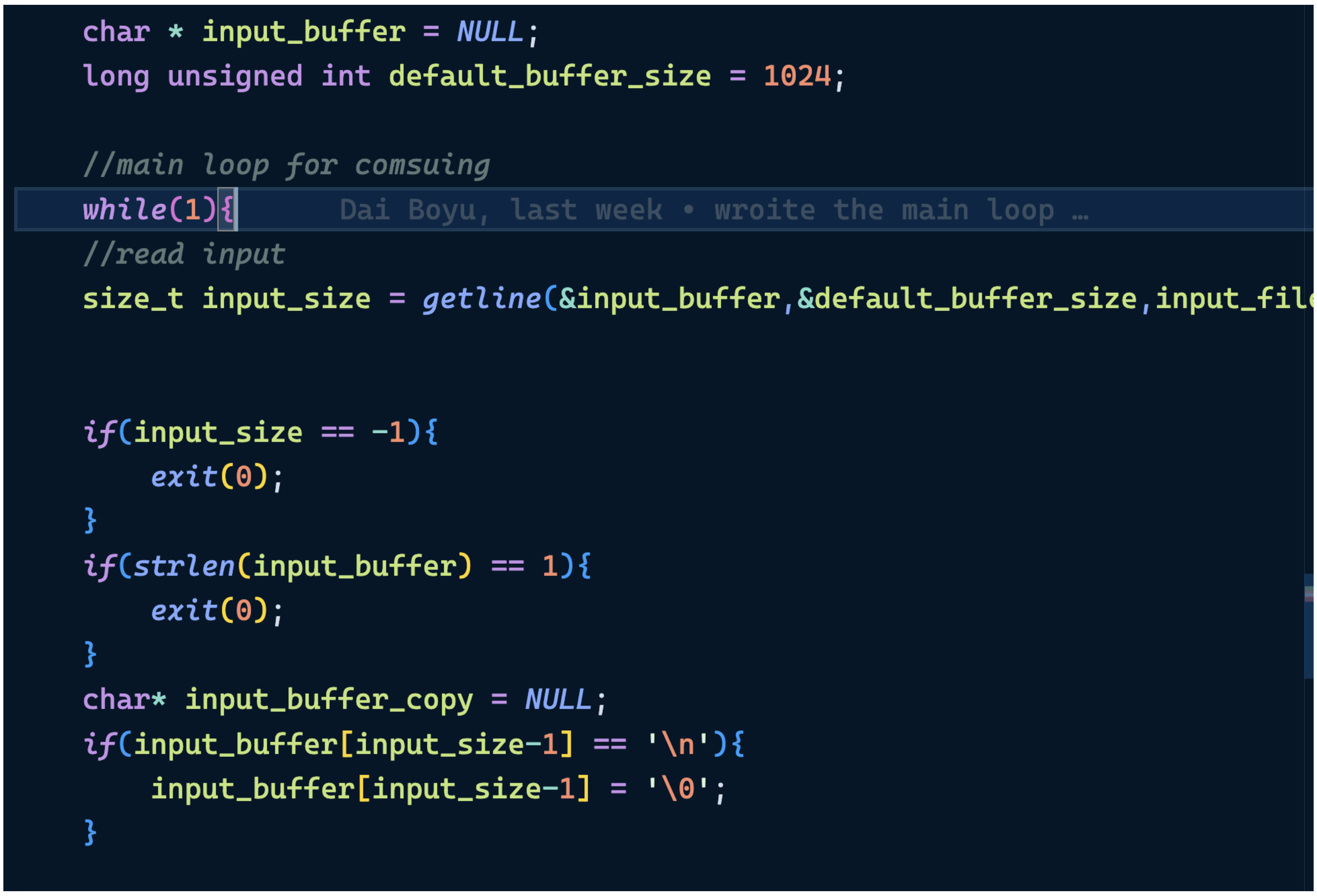
Anubis.c contains the programme's main function and is responsible for *setup initial data structures*, *reading user input*, and *calling the corresponding function* to process the user input.

Data structures

```
//setup fiel structure for anubis
    //path related structre
path_list* path_list = create_path_list();
if(path_list == -1){
    return 1;
}
path_list = append_path_to_list(path_list,"/bin");

    //setup STDIN
FILE* input_file = NULL;
//args[1] is a file containing input command
  if(argc > 1){
    if(access(argv[1],F_OK|R_OK) == 0){
    //read input form a file
    input_file = fopen(argv[1],"r");
    }else{
        ERROR(errno,"Error opening input file");
        return 1;
    }
}else{
    input_file = stdin;
}
```

It initialises the path_list and sets the stdin of the anubis program. This set of input_file allows the anubis to read commands from a file and run in batch mode.

Command line/File input

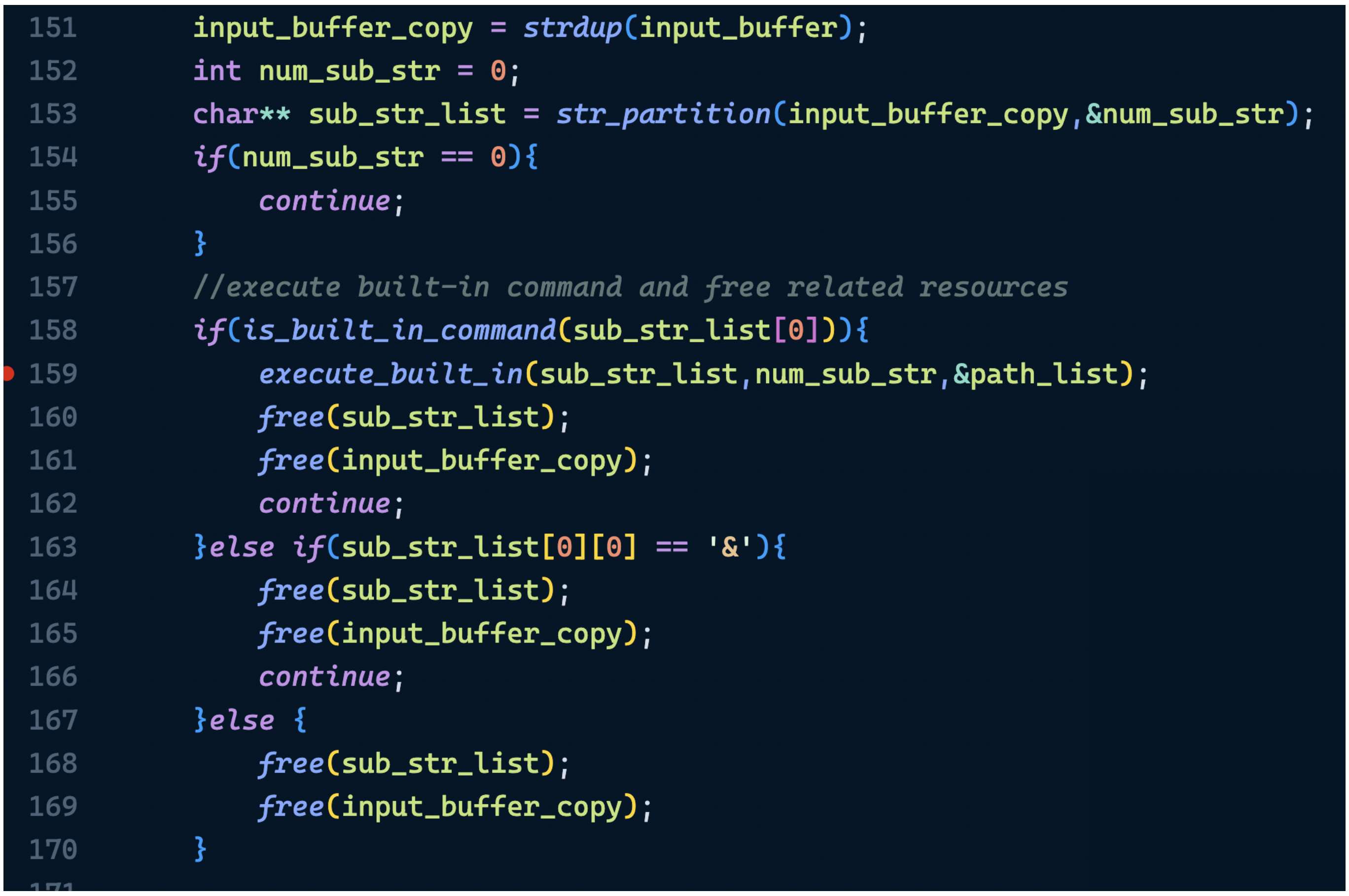


```
    char * input_buffer = NULL;
    long unsigned int default_buffer_size = 1024;

    //main loop for comsuing
    while(1){       Dai Boyu, last week • wroite the main loop …
    //read input
    size_t input_size = getline(&input_buffer,&default_buffer_size,input_fil


    if(input_size == -1){
        exit(0);
    }
    if(strlen(input_buffer) == 1){
        exit(0);
    }
    char* input_buffer_copy = NULL;
    if(input_buffer[input_size-1] == '\n'){
        input_buffer[input_size-1] = '\0';
    }
```

As shown above, the anubis program uses a 'getline()' inside the while loop to consume all the input in the stdin or input file. The program exits when there are *no more lines to read, receiving an empty string or EOF symbol*

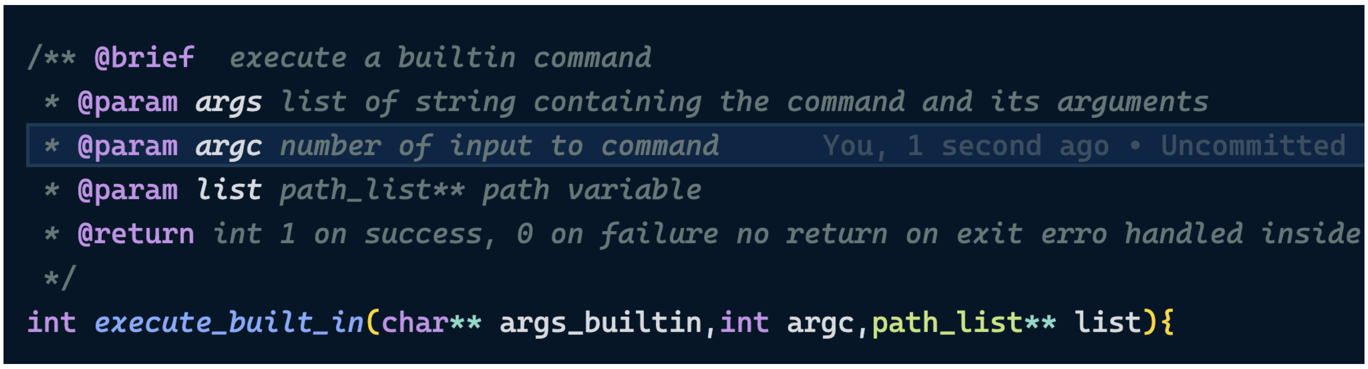
Built-in Command Execution

After reading the input line, the anubis program will check for built-in commands and attempt to execute them. (Consider not using the same name as the built-in command as file name/programme name as that would confuse the program.)



```c
        input_buffer_copy = strdup(input_buffer);
        int num_sub_str = 0;
        char** sub_str_list = str_partition(input_buffer_copy,&num_sub_str);
        if(num_sub_str == 0){
            continue;
        }
        //execute built-in command and free related resources
        if(is_built_in_command(sub_str_list[0])){
            execute_built_in(sub_str_list,num_sub_str,&path_list);
            free(sub_str_list);
            free(input_buffer_copy);
            continue;
        }else if(sub_str_list[0][0] == '&'){
            free(sub_str_list);
            free(input_buffer_copy);
            continue;
        }else {
            free(sub_str_list);
            free(input_buffer_copy);
        }
```

After successfully finding the built-in command in the input's first place, commands will be executed with *execute_built_in* function. The documentation of the *execute_built_in* is as follows(errors will be handled inside the function).



```c
/** @brief  execute a builtin command
 * @param args list of string containing the command and its arguments
 * @param argc number of input to command      You, 1 second ago • Uncommitted
 * @param list path_list** path variable
 * @return int 1 on success, 0 on failure no return on exit erro handled inside
 */
int execute_built_in(char** args_builtin,int argc,path_list** list){
```

Tokenising

After checking for built-in commands, the anubis program will view the input as a line of command that needs to be processed. The first step is tokenising the input string.

```
//command is an input, tokennise,parse and execute
int error_parsing = 0;
Node* tokens = tokenise_str(input_buffer);
if(tokens == NULL){//WTF just skipped
    ERROR(EINVAL,"Error tokenising input, NULL return");
    continue;
}
```

For further details about tokenising, please check tokerniser.c

Parsing

After the call to tokenise the string, the main would parse the command into an expression that allows precedence between symbols to be processd

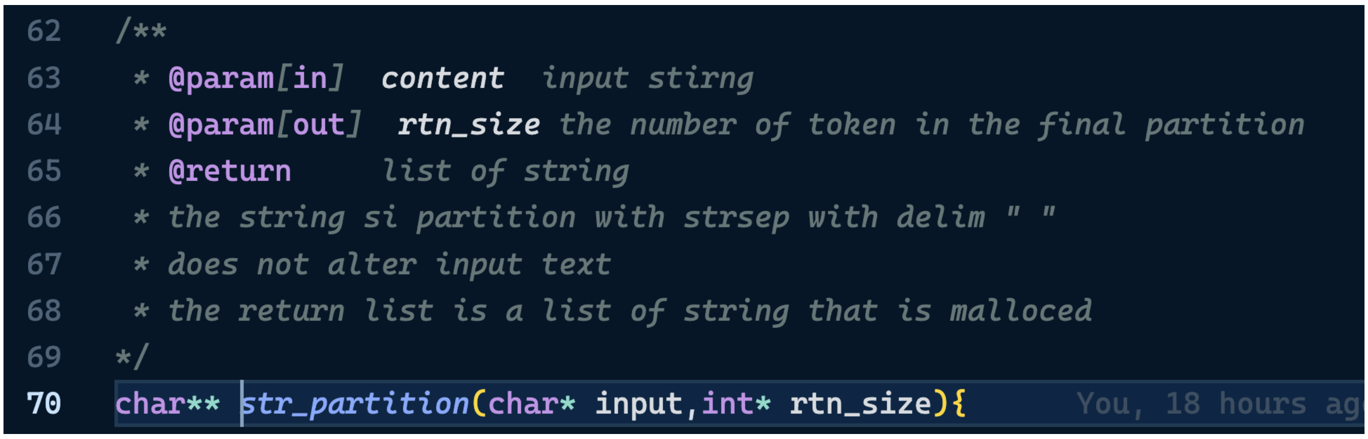
```
//print_token_list(tokens);
//parsing input
//case 11 make problem here
parallel_exp* parsed_exp = parse_parallel(tokens,&error_parsing);
if(error_parsing !=0){
    ERROR(EINVAL,"Error parsing input");
    continue;
}
```

For further details about the parsing process, please check parser.c

Tokerniser.c

This file contains code related to the tokenisation of commands and strings.

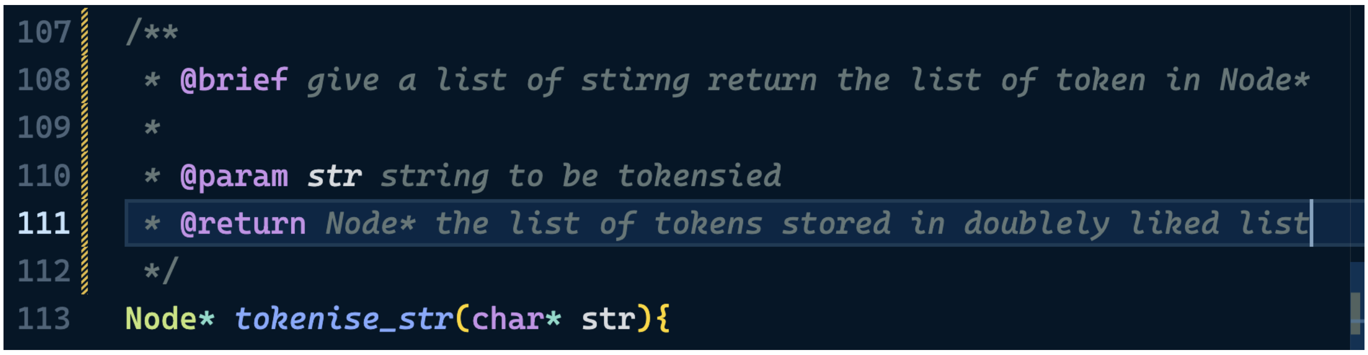
Partitioning String



```
/**
 * @param[in]  content  input stirng
 * @param[out]  rtn_size the number of token in the final partition
 * @return      list of string
 * the string si partition with strsep with delim " "
 * does not alter input text
 * the return list is a list of string that is malloced
*/
char** str_partition(char* input,int* rtn_size){       You, 18 hours ag
```

The string pattern is the first step of tokenising the string. The str_partition function divides the string by empty spaces and the special symbols so that they can be tokenised into tokens.

Tokenise_str



```c
/**
 * @brief give a list of stirng return the list of token in Node*
 *
 * @param str string to be tokensied
 * @return Node* the list of tokens stored in doublely liked list
 */
Node* tokenise_str(char* str){
```

The tokenise_str utilises the partition string function. Construct *Token* out of the substring and return in *Node*(which is a linked list defined in utils.c)

Parser.c

As mentioned, the anubis assumes the incoming command is in the form of the below grammar.

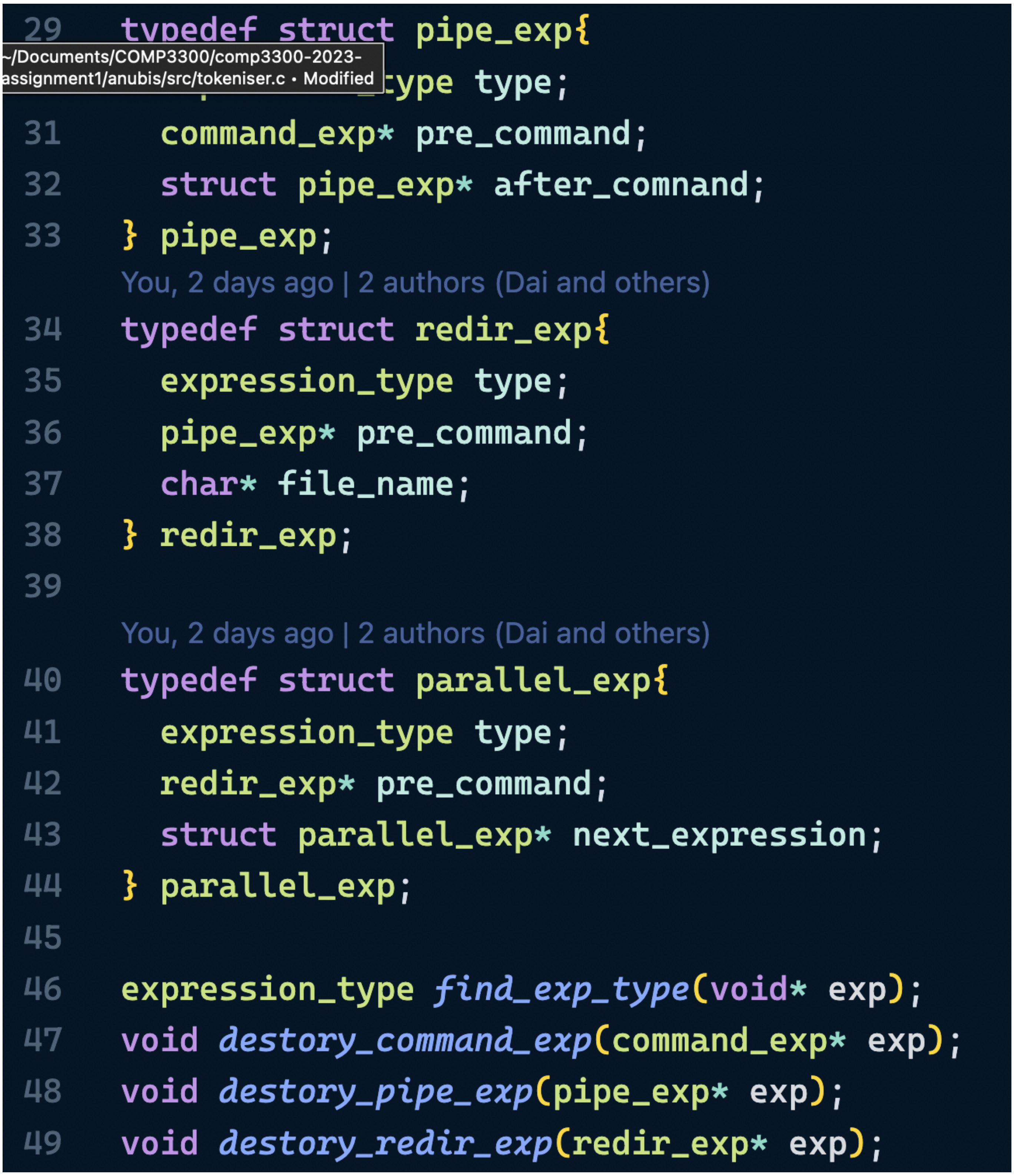
command_exp := name_of_bin and arguments

redir_exp := pipe_exp | pipe_exp > file_name

pipe_exp := command_exp / command_exp | pipe_exp

parallel_exp := redir_exp | redir_exp & parallel_exp

Parser.c contains functions and data structures related to parsing a linked list of tokens.



```
~/Documents/COMP3300/comp3300-2023-
assignment1/anubis/src/tokeniser.c • Modified
```

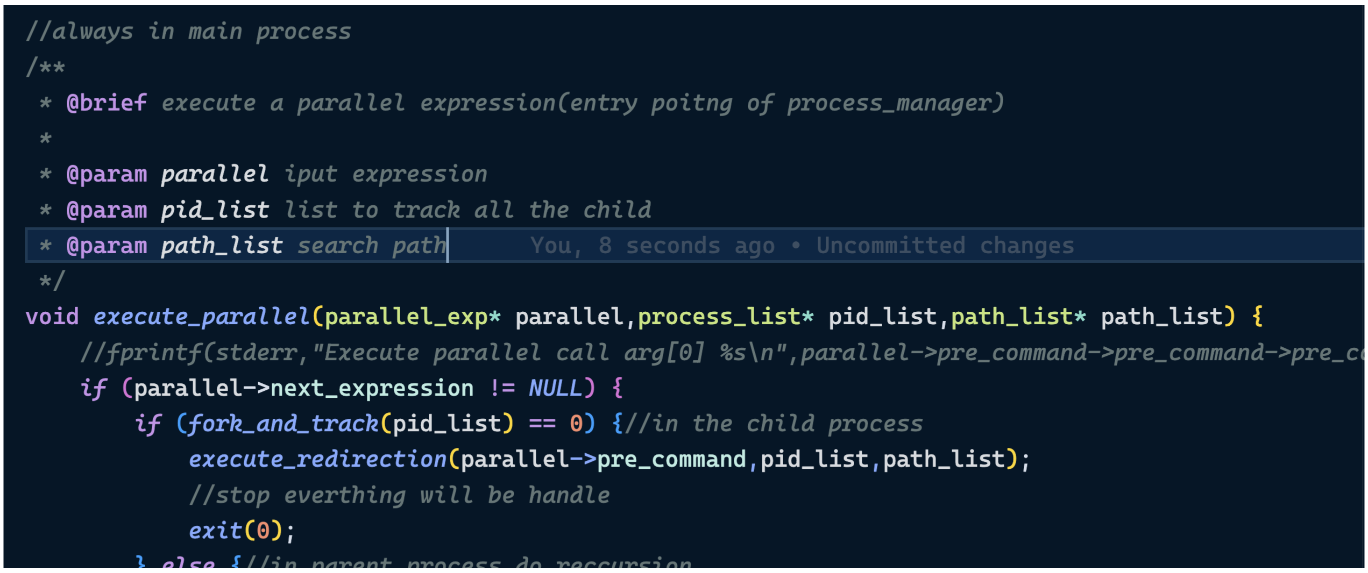
```c
typedef struct pipe_exp{
             type type;
    command_exp* pre_command;
    struct pipe_exp* after_comnand;
} pipe_exp;
You, 2 days ago | 2 authors (Dai and others)
typedef struct redir_exp{
    expression_type type;
    pipe_exp* pre_command;
    char* file_name;
} redir_exp;

You, 2 days ago | 2 authors (Dai and others)
typedef struct parallel_exp{
    expression_type type;
    redir_exp* pre_command;
    struct parallel_exp* next_expression;
} parallel_exp;

expression_type find_exp_type(void* exp);
void destory_command_exp(command_exp* exp);
void destory_pipe_exp(pipe_exp* exp);
void destory_redir_exp(redir_exp* exp);
```

In parsing a command, *parse_parallel* is executed on the list of input tokens. It will let *parse_redir* consume tokens until encountering a "&". Parse rider saves the result of parse_redir and, based on the preceding symbole, constructs next *parallel_exp*.

The *parse_redir* and *parse_pipe* adapt similar structure to the *parse_paraallel*

process_manager.c

The process_manager contains functions and data structures related to forking and executing expressions.



```
//always in main process
/**
 * @brief execute a parallel expression(entry poitng of process_manager)
 *
 * @param parallel iput expression
 * @param pid_list list to track all the child
 * @param path_list search path        You, 8 seconds ago • Uncommitted changes
 */
void execute_parallel(parallel_exp* parallel,process_list* pid_list,path_list* path_list) {
    //fprintf(stderr,"Execute parallel call arg[0] %s\n",parallel->pre_command->pre_command->pre_c
    if (parallel->next_expression != NULL) {
        if (fork_and_track(pid_list) == 0) {//in the child process
            execute_redirection(parallel->pre_command,pid_list,path_list);
            //stop everthing will be handle
            exit(0);
        } else {//in parent process do reccursion
```

Similar to the parse, the execution of the command is also in a hierarchical structure.

Recall that a *parallel_exp* contains a redir_exp or an optional link to the following *parallel_exp*. In executing, the process would call *fork()* for each redir_exp and let the child process call *execute_redirection()* to execute all commands.