Report, Assignment 1

Name: Giorgos Nikolaou
SDI: 1115-202000154

1 General Approach

For this assignment, I decided to use the Visitor Pattern to simplify the development process. Although it required significant effort initially due to the various components that needed to be developed prior to focusing on the core functionality of the shell, the approach resulted in clean and well-structured code. As a result, implementing the shell itself was relatively straightforward.

1.1 Scanner

A scanner is a program that takes an input and produces a series of tokens that are typically used by a parser.

The main challenge for the scanner is identifying words accurately. A word is a sequence of characters that are not white spaces or special characters (`>`, `<`, `|`, `&`, `;`, `*`, `?`, `$`, `"`, `'`). Words can include these characters if they are escaped or if they are enclosed in single or double quotes.

The scanner handles unescaped wild characters ($*, ?$) by using the `glob` function. Once a word is identified, the function is called with the word as an argument, and produces a list of words that correspond to files that can be generated by considering the wild characters. If the list is empty, only the original word is returned; otherwise, each produced word is returned.

The special character `$` is used to retrieve environment variables. Anything matching `[a-zA-Z_][a-zA-Z_0-9]`* after the symbol is treated as the name of the environment variable the user is requesting. If the variable does not exist, an empty string is returned according to the standard convention. Environment variables do not terminate a word, they are simply replaced by their value before continuing.

Strings are sequences of characters enclosed in single or double quotes. A string is treated as a single token and can be used to ignore unwanted special characters easily. They, too, don't terminate a word. Both types of quotes can be used inside the string, provided they are escaped. Every opening quote must have a corresponding closing quote.

1.2 Parser

A parser is a program that takes a list of tokens and determines whether the sequence conforms to its grammar. It then creates the Abstract Syntax Tree that will be used by the visitors.

The shell commands we are interested in can be identified using a simple (almost) `LL(1)` grammar, which can be easily implemented using a recursive descent parser. The grammar is provided inside of `shell.grammar`.

1.3 Visitor

Following the classic Visitor Pattern,

```
class Visitor {
    virtual void visit(Node* v);
    virtual void visit(Commands* m);
    virtual void visit(Pipe* s);
                ⋮
}

class Shell : public Visitor {
    public:
    void visit(Node* node) { ... }
    void visit(Pipe* pipe) {
                    ⋮
        pipe->left->accept(this);
                    ⋮
    }
    void visit(Commands* com) { ... }
                ⋮
}

⋮

class Pipe : public Node {
    public:
    void accept(Visitor* v) { v->visit(this); }
                    ⋮
}
```

we can finally develop an interpreter for shell commands. The idea behind the visitor pattern, is that by determining the behavior for the different possible components, we can recursively handle complex commands while keeping the code clean and easy to understand.

2 Implementation Details

2.1 Wild-characters, Environment Variables & Assumptions

As described in 1.1, wild-characters and environment variables are handled during the tokeniziation of the input, eliminating the need for elaborate handling later.
The following assumptions have been made:

- Input cannot include a new line
- Every opening quote that is not taken "as-is" (if it is inside of a string or is escaped), must have the corresponding closing quote. Otherwise, we throw a parse error.
- Commands that are either suspended or run on the background when the shell closes, are collected and killed.
- It is possible to recursively run our shell only by running the executable directly.

2.2 Process Groups & Background Commands

When the shell is created, it creates a new process group for itself and sets it to be the foreground process group. This is done for two reasons. Firstly, it allows the main program (our shell) to receive the signals correctly even when using a makefile. Secondly, it later lets as handle children and signals with ease, not having to handle edge cases manually.

For non-piped commands, when the child is forked it creates a new process group for itself before continuing, while the parent (if it is not a background command) brings the new process group to the foreground, giving it control of the terminal and waits for it to finish.

For piped commands, when the first child is forked it creates a new process group for itself before continuing, acting as the leader. Every other command, sets its process group id to that of the leader, allowing to handle the pipe as a whole. If pipe is not a background command, the parent brings the process group of the leader to the foreground, giving it control of the terminal and waits for exactly two of its children to finish.

When the execution of a command is finished, the shell regains control of the terminal and continues. By organising the commands in process groups, we make sure that signals are only received by the correct processes.

2.3 Signals

Having organised the processes in groups, signal handling became extremely simple to implement. Specifically:

- The shell ignores the following signals
  - `SIGINT` and `SIGTSTP`, i.e. `Ctrl+C` and `Ctrl+Z` respectively, thus not stopping if they are given while it's on the foreground.
  - `SIGCHLD`. By explicitly using `SIG_IGN`, the behavior of the process when a child exits, is to reap it immediately rather than keep it as a zombie and let the process call a wait to do so.
  - `SIGTTIN` and `SIGTTOU` to correctly handle io when running in the background. This, as well as the behavior for `SIGCHLD`, is inherited by any children forked, which allows them to have the correct behavior without needing any changes.
- Every process that is forked in order to execute a command, we revert the disposition of `SIGINT` and `SIGTSTP` to `SIG_DFL`. Therefore if they are in the foreground and the user gives any of the two signals, the process responds with the correct behavior. For pipes, since all sub-processes are in the same process group, they all receive the signal at the same time, stopping/suspending execution as a unit.

2.4 Redirections

Redirections are handled by just changing `stdin` and `stdout` for the process. This is done iteratively, essentially only the last file is used, although on output redirection all files are created if they do not already exist.

2.5 Commands

There is a set of special commands that have to be handled manually:

- `exit`: Closes the shell and exits. Allows arguments, does not take them into account.
- `history <num>?`: Allows at most one argument that must be a positive integer greater than 1. If the argument does not exist, prints the last 20 commands entered, otherwise executes the corresponding one.
- `cd`: Changes the working directory. Allows arguments, does not take them into account
- `createalias` and `destroyalias`: Create or destroy an alias. When creating one, there should be exactly two arguments, while when destroy one there should be exactly two.

Every other command is executed the same way, we fork a child, set the redirections and call `execvp`. If it is a background command the parent just does not wait for the child to finish. If it is not, the parent brings it to the foreground and waits for it to finish.

2.6 Pipes

Recursively fork a child for the left command (a `SubPipe`) and one for the right command (a `SubPipe` or possibly another `Pipe`). Before proceeding with the execution, we change the input and output of the process to the corresponding pipe end. This follows the standard bash convention, that is if we have `cat nums.txt | sort < nums2.txt`, `sort` would get `nums2.txt` as input and `cat` might produce a broken pipe error.

3 Array of implemented functionalities

	General Notes	Implemented	Partially Implemented	Not Implemented
Shell	Basic shell functionality, entering and using a command	✓		
	Return and printing of results	✓		
Redirections	Single Redirection (>or <)	✓		
	Multiple Redirection, both input and output multiple times	✓		
	Append Redirection	✓		
Pipes	Simple Pipe	✓		
	Combination with redirections	✓		
Background Execution	Execution of commands on the background	✓		
	Execution of multiple commands in a single line	✓		
	Return completion status back to the shell	✓		
Wild-Chars	* support	✓		
	? support	✓		
Aliases	Creation and usage of aliases	✓		
	Destruction of an alias	✓		
History	Storing and printing history	✓		
	Call command from history	✓		
Signals	Control+C kills the process	✓		
	Control+C doesn't affect the shell	✓		
	Control+Z suspends the process	✓		
	Control+Z doesn't affect the shell	✓		