P1 - Bash-like Shell

Aakash 2018B4A70887P
Sourabh 2018B3A70815P

Contents

Code Details

Broad View

A bash like shell that used system-call wrappers to execute user input commands.
We use execv() call to run the user input commands.

Compilation

Makefile :Used to compile the code. There are 2 options -

1. Compile : This option can be used to compile the code. This is also the default option for running make
2. Clean : This option can be used to clean all files that are generated by makefile.

Makefile will not create any new folders.

Execution

To run the shell execute the following command
./shell.o

This will run a bash-like shell that runs commands that the user inputs.
To exit this shell, press “exit”.

Structure :

Parsing
As a first step when the user inputs the command, we parse the given command.
We have created grammar rules that accepts all valid formats for the input command.
If the input command is syntactically incorrect, we report it appropriately.
We follow the general methodology followed for compilers for parsing the input.
The following files help us perform the parsing -
AST.h

AST.cpp
DFA.txt
Grammar.txt
Lexer.h
Lexer.cpp
Parser.h
Parser.cpp

Parser.cpp

1. computeNullables()
   Given the grammar rules, this function can be called to find out the set of nullable non terminals
2. computeFirstSets()
   Given the grammar rules, this function can be called to find out the first sets of non terminals
3. computerFollowSets()
   Given the grammar rules, this function can be called to find out the follow sets of non terminals
4. computeParseTable()
   Computes the parse table for the given set of grammar rules
5. loadParser()
   Called to initialize the parser
6. _pop() and printStack()
   Helper functions in the parser that can be called to pop from the stack and print the stack state respectively.

Lexer.cpp

1. loadDFA()
   Called to initialize the DFA for accepting or rejecting the input command
2. getNextToken()
   This function is called by the parser to receive tokens in a serial order according to the user input.

Structures

Once we obtain a syntactically correct command, we then proceed to execute the commands. The shell has the ability to selectively stop or resume processes. For this the shell associates a job with each pipeline. So each job is a pipeline of multiple processes.
We create class for implementing jobs and processes.
This has been written in the following files -
Job.h
Job.cpp
Process.h
Process.cpp

When the shell enables job control, it must make sure that it has to ignore all the job control stop signals in order to ensure it does not stop itself accidentally.

We use fork and exec functions to create to create the processes for each subcommand in the process group/
All the processes in the process group are a child of the shell process. As each process if forked, it puts itself in the new process group by using the setpgid() call. Each process then resets the signal actions so as to no inherit the parent(shell's) signal actions because the shell ignores all job control signals which should not be done by the child processes.
Inside each process, the file descriptors are set appropriately. The standard input and output channels must be redirected appropriately and then we call the execv() call to execute the command.
When we launch a job, all the processes belonging to the job are run in parallel.
Whenever a background process is stopped or terminated, the shell reports it to the user.

Job.cpp

1. markProcessStatus(pid_t pid, int status)
   This is called to update the status of each process in the process with process id pid.
2. waitForJob()
   This is called to wait for a job to report
3. putInBackground(cont)
   This is called to put a job in background. If cont is set, then SIGCONT signal is sent to the process group with group id equal to the Job's pgid
4. isStopped()
   This is called to check if a job has been stopped or not
5. isCompleted()
   This is called to check if the job has completed its execution or not.
6. launch()
   This creates a daemon process if the user has so mentioned, and then launches each of the job's processes.

Process.cpp

1. launch()
   This launches the process. It sets the file descriptor appropriately, computes the executable path, reports error if any, and then executes the command which has been assigned to the process.
2. char** splitString(char* input, int&size, char delim)
   This is a helper function that splits the input string with respect to the delimiter and also sets the size of the final output array of tokens as the total number of tokens that has created.

3. char* getExecutablePath(const char* execName)
   This is also a helper function that is called to find the executable path of the given command.

Shell.cpp

1. getInstance()
   Since there is only one shell, the shell is implemented as a Singleton class. This function can be used to return the singleton instance of the shell.
2. executeJob(const car* input)
   Given a input from the user, this function can be called to execute the job.
3. initialize()
   This function is called to initialize the shell variables - the group id, signals that have to be ignored for job control, process id of the shell.
4. printPrompt()
   This is called each time the shell has completed executing the task. Prints the current working directory and shows that it is ready for executing a new command.

Execution Flow

1. The shell is initialized.
2. Until user exits
   a. Print the prompt
   b. Wait for input
   c. Read input
   d. Parse input
   e. Set as background process, daemonize
   f. Set input output redirections
   g. Run the commands
   h. Display output