Chapter 5. Writing Your Own Shell

You really understand something until you program it.
--GRR

Introduction

Last chapter covered how to use a shell program using UNIX commands. The shell is a program that interacts with the user through a terminal or takes the input from a file and executes a sequence of commands that are passed to the Operating System. In this chapter you are going to learn how to write your own shell program.

Shell Programs

A shell program is an application that allows interacting with the computer. In a shell the user can run programs and also redirect the input to come from a file and output to come from a file. Shells also provide programming constructions such as if, for, while, functions, variables etc. Additionally, shell programs offer features such as line editing, history, file completion, wildcards, environment variable expansion, and programing constructions. Here is a list of the most popular shell programs in UNIX:

sh	Shell Program. The original shell program in UNIX.
csh	C Shell. An improved version of sh.
tcsh	A version of Csh that has line editing.
ksh	Korn Shell. The father of all advanced shells.
bash	The GNU shell. Takes the best of all shell programs. It is currently the most common shell program.

In addition to command-line shells, there are also Graphical Shells such as the Windows Desktop, MacOS Finder, or Linux Gnome and KDE that simplify the use of computers for most of the users. However, these graphical shells are not substitute to command line shells for power users who want to execute complex sequences of commands repeatedly or with parameters not available in the friendly, but limited graphical dialogs and controls.

Parts of a Shell Program

The shell implementation is divided into three parts: **The Parser**, **The Executor**, and **Shell Subsystems.**

The Parser

The Parser is the software component that reads the command line such as “ls -al” and puts it into a data structure called ***Command Table*** that will store the commands that will be executed.

The Executor

The executor will take the command table generated by the parser and for every SimpleCommand in the array it will create a new process. It will also if necessary create pipes to communicate the output of one process to the input of the next one. Additionally, it will redirect the standard input, standard output, and standard error if there are any redirections.

The figure below shows a command line “A | B | C | D”. If there is a redirection such as “**< *infile***” detected by the parser, the input of the first SimpleCommand A is redirected from ***infile***. If there is an output redirection such as “**> *outfile***”, it redirects the output of the last SimpleCommand (D) to ***outfile***.

If there is a redirection to errfile such as “**>& errfile**” the stderr of all SimpleCommand processes will be redirected to **errfile**.

Shell Subsystems

Other subsystems that complete your shell are:

- Environment Variables: Expressions of the form ${VAR} are expanded with the corresponding environment variable. Also the shell should be able to set, expand and print environment vars.
- Wildcards: Arguments of the form a*a are expanded to all the files that match them in the local directory and in multiple directories .
- Subshells: Arguments between `` (backticks) are executed and the output is sent as input to the shell.

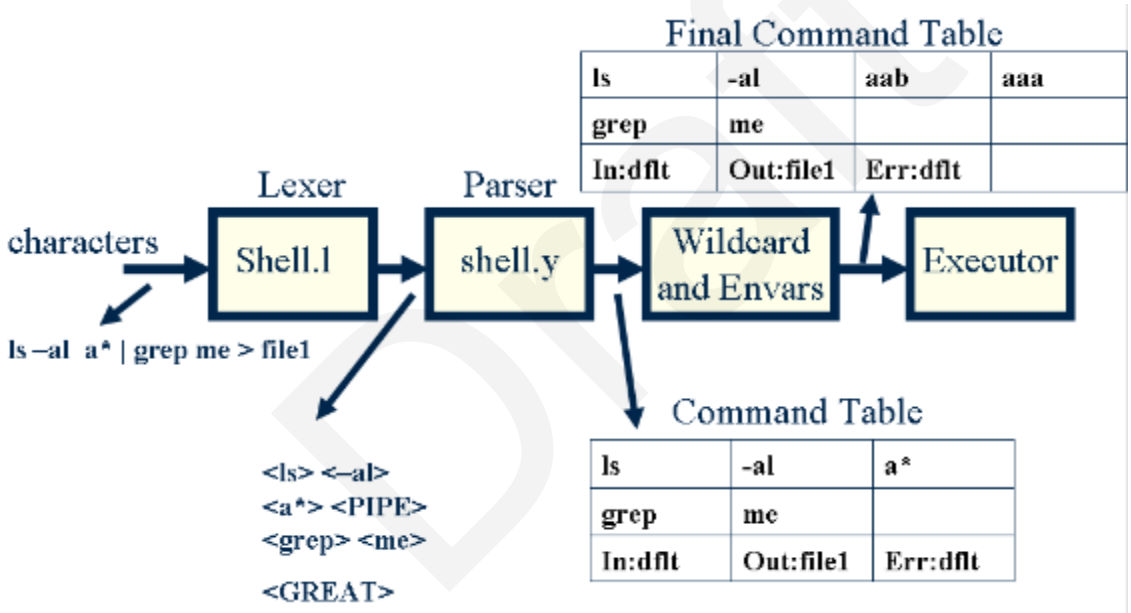
We highly recommend that you implement your own shell following the steps in this chapter. Implementing your own shell will give you a very good understanding of how the shell interpreter applications and the operating system interact. Also, it will be a good project to show during your job interview to future employers.

Using Lex and Yacc to implement the Parser

You will use two UNIX tools to implement your parser: Lex and Yacc. These tools are used to implement compilers, interpreters, and preprocessors. You do not need to know compiler theory to use these tools. Everything you need to know about these tools will be explained in this chapter.

A parser is divided into two parts: a **Lexical Analyzer** or **Lexer** takes the input characters and puts the characters together into words called **tokens**, and a **Parser** that processes the tokens according to a grammar and build the command table.

Here is a diagram of the Shell with the Lexer, the Parser and the other components.



The ***tokens*** are described in a file **shell.l** using regular expressions. The file **shell.l** is processed with a program called **lex** that generates the lexical analyzer.

The grammar rules used by the parser are described in a file called **shell.y** using syntax expressions we describe below. **shell.y** is processed with a program called **yacc** that generates a parser program. Both lex and yacc are standard commands in UNIX. These commands could be used to implement very complex compilers. For the shell we will use a subset of Lex and Yacc to build the command table needed by the shell.

You need to implement the below grammar in ***shell.l*** and ***shell.y*** to make our parser interpret the command lines and provide our executor with the correct information.

```
cmd [arg]* [ | cmd [arg]* ]*
        [ [> filename] [< filename] [ >& filename] [>> filename] [>>& filename] ]* [&]
```

Fig 4: Shell Grammar in Backus-Naur Form

This grammar is written in a format called *"Backus-Naur Form"*. For example **cmd [arg]*** means a command, ***cmd***, followed by 0 or more arguments, ***arg***. The expression **[| cmd [arg]*]*** represents the optional pipe subcommands where there might be 0 or more of them. The expression **[>filename]** means that there might be 0 or 1 **>filename** redirections. The ***[&]*** at the end means that the ***&*** character is optional.

Examples of commands accepted by this grammar are:

ls –al
ls –al > out
ls –al | sort >& out
awk –f x.awk | sort –u < infile > outfile &

The Command Table

The ***Command Table*** is an array of ***SimpleCommand*** structs. A ***SimpleCommand*** *struct* contains members for the command and arguments of a single entry in the pipeline. The parser will look also at the command line and determine if there is any input or output redirection based on symbols present in the command (i.e. < infile, or > outfile).

Here is an example of a command and the ***Command Table*** it generates:

command

ls -al | grep me > file1

Command Table

SimpleCommmand array:

0:	**ls**	**-al**	**NULL**
1:	**grep**	**me**	**NULL**

IO Redirection:

in: default	**out: file1**	**err: default**

To represent the command table we will use the following classes: **Command** and **SimpleCommand**.

```
// Command Data Structure

// Describes a simple command and arguments
struct SimpleCommand {
        // Available space for arguments currently preallocated
        int _numberOfAvailableArguments;

        // Number of arguments
        int _numberOfArguments;

        // Array of arguments
        char ** _arguments;

        SimpleCommand();
        void insertArgument( char * argument );
};

// Describes a complete command with the multiple pipes if any
// and input/output redirection if any.
struct Command {
        int _numberOfAvailableSimpleCommands;
        int _numberOfSimpleCommands;
        SimpleCommand ** _simpleCommands;
        char * _outFile;
        char * _inputFile;
        char * _errFile;
        int _background;

        void prompt();
        void print();
        void execute();
        void clear();

        Command();
        void insertSimpleCommand( SimpleCommand * simpleCommand );

        static Command _currentCommand;
        static SimpleCommand *_currentSimpleCommand;
};
```

The constructor **SimpleCommand::SimpleCommand** constructs a simple empty command. The method **SimpleCommand::insertArgument(char * argument)** inserts a new argument into the SimpleCommand and enlarges the _arguments array if necessary. It also makes sure that the last element is NULL since that is required for the exec() system call.

The constructor **Command::Command**() constructs and empty command that will be populated with the method Command::**`insertSimpleCommand( SimpleCommand * simpleCommand)`**. insertSimpleCommand also enlarges the array _simpleCommands if necessary. The variables _outFile, _inputFile, _errFile will be NULL if no redirection was done, or the name of the file they are being redirected to.

The variables **`_currentCommand`** and **`_currentCommand`** are static variables, that is there is only one for the whole class. These variables are used to build the Command and Simple command during the parsing of the command.

The Command and SimpleCommand classes implement the main data structure we will use in the shell.

Implementing the Lexical Analyzer

The Lexical analyzer separates input into tokens. It will read the characters one by one from the standard input. and form a token that will be passed to the parser. The lexical analyzer uses a file ***shell.l*** that contains regular expressions describing each of the tokens. The lexer will read the input character by character and it will try to match the input with each of the regular expressions. When a string in the input matches one of the regular expressions, it will execute the code {...} at the right of the regular expression. The following is a simplified version of shell.l that your shell will use:

```
/*
 * shell.l: simple lexical analyzer for the shell.
 */

%{

#include <string.h>
#include "y.tab.h"

%}

%%

\n      {
                return NEWLINE;
```

```
        }

[ \t]   {
                /* Discard spaces and tabs */
        }

">"     {
                return GREAT;
        }

"<"     {
                return LESS;
        }

">>"     {
                return GREATGREAT;
        }

">&"     {
                return GREATAMPERSAND;
        }

"|"    {
                return PIPE;
        }

"&"    {
                return AMPERSAND;
        }


[^ \t\n][^ \t\n]*  {
                /* Assume that file names have only alpha chars */
                yylval.string_val = strdup(yytext);
                return WORD;
        }

/* Add more tokens here */

.       {
                /* Invalid character in input */
                return NOTOKEN;
        }

%%
```

The file ***shell.l*** is passed through ***lex*** to generate a C file called ***lex.yy.c.*** This file implements the scanner that the parser will use to translate characters into tokens.

Here is the command used to run lex.

```
bash% lex shell.l
bash% ls
lex.yy.c
```

The file ***lex.yy.c*** is a C file that implements the lexer to separate the tokens described in ***shell.l***.

There are two parts in shell.l. The top part looks like this:

```
%{
#include <string.h>
#include "y.tab.h"
%}
```

This is is a portion that will be inserted at the top of the file ***lex.yy.c*** directly without modification that includes header files and variable definitions that you will use in the scanner. That is where you can declare variables you will use in your lexer.

The second portion delimited by ***%%*** looks like this:

```
%%
\n     {
            return NEWLINE;
       }
[ \t]  {
            /* Discard spaces and tabs */
       }
">"    {
            return GREAT;
       }
[^ \t\n][^ \t\n]*  {
            /* Assume that file names have only alpha chars */
            yylval.string_val = strdup(yytext);
            return WORD;
       }
%%
```

This portion contains the regular expressions that define the tokens formed by taking the characters from standard input. Once a token is formed, it will be returned, or in some cases discarded. Each rule that defines a token has also two parts:

regular-expression { action }

E.g.

```
\n     {
            return NEWLINE;
       }
```

The first part is a regular expression that describes the token that we expect to match. The action is a piece of C code that the programmer adds that is executed once the token matches the regular expression. In the example above when the character newline is found, lex will return the constant ***NEWLINE***. We will describe later where the ***NEWLINE*** constants are defined.

Here is a more complex token that describes a ***WORD***. A ***WORD*** can be an argument for a command or the command itself.

```
[^ \t\n][^ \t\n]*  {
                /* Assume that file names have only alpha chars */
                yylval.string_val = strdup(yytext);
                return WORD;
}
```

The expression in ***[...]*** matches any character that is inside the brackets. The expression ***[^...]*** matches any character that is not inside the brackets. Therefore, ***[^ \t\n][^ \t\n]**** describes a token that starts with a character that is not a space, tab or newline and is succeeded by zero or more characters that are not spaces, tabs, or newlines. The token matched is in a variable called ***yytext***. Once a word is matched, a duplicate of the token matched is assigned to ***yylval.string_val*** in the following statement:

```
yylval.string_val = strdup(yytext);
```

this is the way the value of the token is passed to the parser. Finally, the constant ***WORD*** is returned to the parser.

Adding new tokens to shell.l

The ***shell.l*** described above currently supports a reduced number of tokens. As the first step in developing your shell you will need to add more tokens to the new grammar that are not currently in ***shell.l*** . See the grammar in ***Figure 4*** to see what tokens are missing and need to be added to ***shell.l*** Here are some of these tokens:

">>" { return GREATGREAT; }
“|” { return PIPE;}
“&” { return AMPERSAND}
Etc.

Adding the new tokens to shell.y

You will add the token names you created in the previous step into ***shell.y*** in the %token section:

%token NOTOKEN, GREAT, NEWLINE, WORD, GREATGREAT, PIPE, AMPERSAND etc

Completing the grammar

You need to add more rules to shell.y to complete the grammar of the shell. The following figure separates the syntax of the shell into different parts that will be used to build the grammar.

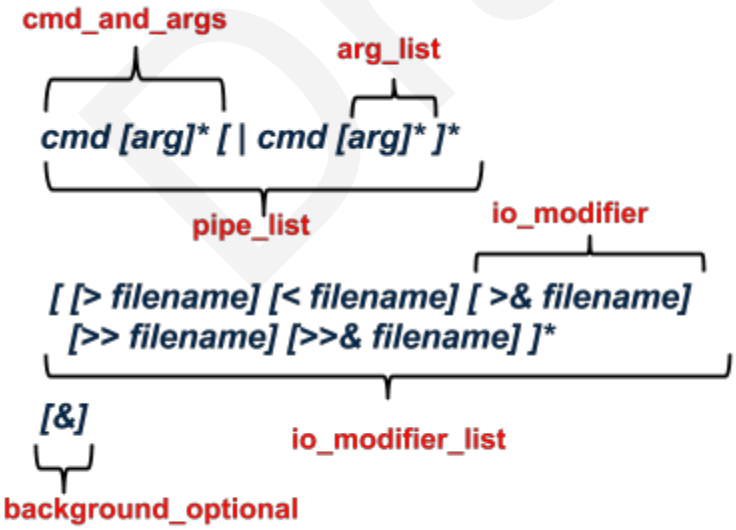


Figure 3. Shell Grammar labeld with the different parts.

Here is the grammar formed using the labeling defined above:

```
goal: command_list;
```

```
arg_list:
        arg_list WORD
        | /*empty*/
                ;
cmd_and_args:
        WORD arg_list
                ;
pipe_list:
        pipe_list PIPE cmd_and_args
        | cmd_and_args
        ;

io_modifier:
        GREATGREAT Word
         | GREAT Word
         | GREATGREATAMPERSAND Word
         | GREATAMPERSAND Word
         | LESS Word
        ;
io_modifier_list:
        io_modifier_list io_modifier
                | /*empty*/
                ;
background_optional:
                AMPERSAND
                | /*empty*/
                ;
command_line:
         pipe_list io_modifier_list background_opt NEWLINE
        | NEWLINE /*accept empty cmd line*/
        | error NEWLINE{yyerrok;}
                 /*error recovery*/

command_list :
                 command_list command_line
                ;/* command loop*/
```

The grammar above implements the command loop in the grammar itself.

The error token is a special token used for error recovery. error will parse all tokens until a token that is known is found like <NEWLINE>. ***yerrok*** tells parser that the error was recovered.

The parser takes the tokens generated by the lexical analyzer and checks if they follow the syntax described by the grammar rules in ***shell.y***. While checking if the input command line follows the syntax, the parser will execute actions or pieces of C code that you will insert in between the grammar rules. This pieces of code are called actions and they are delimited by curly braces { action; }.

You need to add actions {…} in the grammar to fill up the command table.

Example:

```
 arg_list:
    arg_list WORD { currSimpleCmd->insertArg($2); }
    | /*empty*/
       ;
```

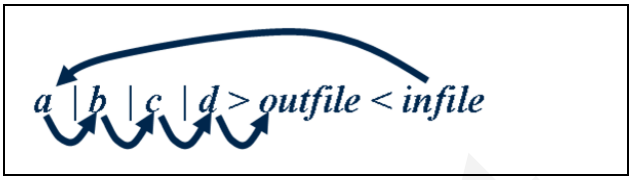
Creating Processes in Your Shell

Start by creating a new process for each command in the pipeline and making the parent wait for the last command. This will allow running simple commands such as “ls -al”.

```
Command::execute()
{
  int ret;
  for ( int i = 0; i < _numberOfSimpleCommands; i++ ) {
     ret = fork();
     if (ret == 0) {
       //child
       execvp(sCom[i]->_args[0], sCom[i]->_args);
       perror("execvp");
       _exit(1);
     }
     else if (ret < 0) {
       perror("fork");
       return;
     }
     // Parent shell continue
  } // for
  if (!background) {
     // wait for last process
     waitpid(ret, NULL);
  }
}// execute
```

Pipe and Input/Output Redirection in Your Shell

The strategy for your shell is to have the parent process do all the piping and redirection before forking the processes. In this way the children will inherit the redirection. The parent needs to save input/output and restore it at the end. Stderr is the same for all processes



In this figure the process ***a*** sends the ouptut to ***pipe 1***. Then ***b*** reads its input from ***pipe 1*** and sends its output to ***pipe 2*** and so on. The last command ***d*** reads its input from ***pipe 3*** and send its output to ***outfile***. The input from a comes from ***infile***.

The following code show how to implement this redirection. Some error checking was eliminated for simplicity.

```
 1 void Command::execute(){
 2     //save in/out
 3     int tmpin=dup(0);
 4     int tmpout=dup(1);
 5
 6     //set the initial input
 7     int fdin;
 8     if (infile) {
 9       fdin = open(infile,O_READ);
10     }
11     else {
12       // Use default input
13       fdin=dup(tmpin);
14     }
15
16     int ret;
17     int fdout;
18     for(i=0;i<numsimplecommands; i++) {
19       //redirect input
20       dup2(fdin, 0);
21       close(fdin);
22       //setup output
23       if (i == numsimplecommands-1){
24         // Last simple command
25         if(outfile){
```

```
26            fdout=open(outfile,â€¦â€¦);
27          }
28          else {
29            // Use default output
30            fdout=dup(tmpout);
31          }
32        }
33
34        else {
35           // Not last
36           //simple command
37           //create pipe
38           int fdpipe[2];
39           pipe(fdpipe);
40           fdout=fdpipe[1];
41           fdin=fdpipe[0];
42        }// if/else
43
44        // Redirect output
45        dup2(fdout,1);
46        close(fdout);
47
48        // Create child process
49        ret=fork();
50        if(ret==0) {
51          execvp(scmd[i].args[0], scmd[i].args);
52          perror(â€œexecvpâ€);
53          _exit(1);
54        }
55      } //  for
56
57      //restore in/out defaults
58      dup2(tmpin,0);
59      dup2(tmpout,1);
60      close(tmpin);
61      close(tmpout);
62
63      if (!background) {
64        // Wait for last command
65        waitpid(ret, NULL);
66      }
67
68  } // execute
```

The method execute() is the backbone of the shell. It executes the simple commands in a separate process for each command and it performs the redirection.

Lines 3 and 4 save the current stdin and stdout into two new file descriptors using the dup() function. This will allow at the end of execute() to restore the stdin and stdout the way it was

at the beginning of execute(). The reason for this is that stdin and stdout (file descriptors 0 and 1) will be modified in the parent during the execution of the simple commands.

```
 3    int tmpin=dup(0);
 4    int tmpout=dup(1);
```

Lines 6 to 14 check if there is input redirection file in the command table of the form "command < infile". If there is input redirection, then it will open the file in ***infile*** and save it in ***fdin***. Otherwise, if there is no input redirection, it will create a file descriptor that refers to the default input. At the end of this block of instructions ***fdin*** will be a file descriptor that has the input of the command line and that can be closed without affecting the parent shell program.

```
 6    //set the initial input
 7    int fdin;
 8    if (infile) {
 9      fdin = open(infile,O_READ);
10    }
11    else {
12      // Use default input
13      fdin=dup(tmpin);
14    }
```

Line 18 is the ***for*** loop that iterates over all the simple commands in the command table. This ***for*** loop will create a process for every simple command and it will perform the pipe connections.

Line 20 redirects the standard input to come from ***fdin***. After this any read from stdin will come from the file pointed by ***fdin***. In the first iteration, the input of the first simple command will come from ***fdin***. ***fdin*** will be reassigned to a input pipe later in the loop. Line 21 will close ***fdin*** since the file descriptor will no longer be needed. In general it is a good practice to close file descriptors as long as they are not needed since there are only a few available (normally 256 by default) for every process.

```
16    int ret;
17    int fdout;
18    for(i=0;i<numsimplecommands; i++) {
19      //redirect input
20      dup2(fdin, 0);
21      close(fdin);
```

Line 23 checks if this iteration corresponds to the last simple command. If this is the case, it will test in Line 25 if there is a output file redirection of the form “command > outfile” and open ***outfile*** and assign it to ***fdout***. Otherwise, in line 30 it will create a new file descriptor that points to the default input. Lines 23 to 32 will make sure that ***fdout*** is a file descriptor for the output in the last iteration.

```
23      //setup output
23      if (i == numsimplecommands-1){
24        // Last simple command
25        if(outfile){
26          fdout=open(outfile,â€¦â€¦);
27        }
28        else {
29          // Use default output
30          fdout=dup(tmpout);
31        }
32      }
33
34      else {...
```

Lines 34 to 42 are executed for simple commands that are not the last one. For these simple commands, the output will be a pipe and not a file. Lines 38 and 39 create a new pipe. The new pipe. A pipe is a pair of file descriptors communicated through a buffer. Anything that is written in file descriptor fdpipe[1] can be read from fdpipe[0]. IN lines 41 and 42 fdpipe[1] is assigned to fdout and fdpipe[0] is assigned to fdin.

Line 41 `fdin=fdpipe[0]` may be the core of the implementation of pipes since it makes the input fdin of the next simple command in the next iteration to come from fdpipe[0] of the current simple command.

```
34       else {
35          // Not last
36          //simple command
37          //create pipe
38          int fdpipe[2];
39          pipe(fdpipe);
40          fdout=fdpipe[1];
41          fdin=fdpipe[0];
42       }// if/else
43
```

Lines 45 redirect the stdout to go to the file object pointed by fdout. After this line, the stdin and stdout have been redirected to either a file or a pipe. Line 46 closes fdout that is no longer needed.

```
44        // Redirect output
45        dup2(fdout,1);
46        close(fdout);
```

When the shell program is in line 48 the input and output redirections for the current simple command are already set. Line 49 forks a new child process that will inherit the file descriptors 0,1, and 2 that correspond to stdin, stdout, and stderr, that are redirected to either the terminal, a file, or a pipe.

If there is no error in the process creation, line 51 calls the execvp() system call that loads the executable for this simple command. If execvp succeeds it will not return. This is because a new executable image has been loaded in the current process and the memory has been overwritten, so there is nothing to return to.

```
48        // Create child process
49        ret=fork();
50        if(ret==0) {
51          execvp(scmd[i].args[0], scmd[i].args);
52          perror("execvp");
53          _exit(1);
54        }
55      } //  for
```

Line 55 is the end of the for loop that iterates over all the simple commands.

After the for loop executes, all the simple commands are running in their own process and they are communicating using pipes. Since the stdin and stdout of the parent process has been modified during the redirection, line 58 and 59 call dup2 to restore stdin and stdout to the same file object that was saved in tmpin, and tmpout. Otherwise, the shell will obtain the input from the last file the input was redirected to. Finally, lines 60 and 61 close the temporary file descriptors that were used to save the stdin and stdout of the parent shell process.

```
57      //restore in/out defaults
58      dup2(tmpin,0);
59      dup2(tmpout,1);
60      close(tmpin);
61      close(tmpout);
```

If the "&" background character was not set in the command line, it means that the shell parent process should wait for the last child process in the command to finish before printing the shell prompt. If the "&" background character was set it means that the command line will

run asynchronously with the shell so the parent shell process will not wait for the command to finish and it will print the prompt immediately. After this, the execution of the command is done.

```
63      if (!background) {
64        // Wait for last command
65        waitpid(ret, NULL);
66      }
67
68    } // execute
```

The example above does not do standard error redirection(file descriptor 2). The semantics of this shell should be that all simple commands will send the stderr to the same place. The example given above can be modified to support stderr redirection.

Built In Functions

All built-in functions except printenv are executed by the parent process. The reason for this is that we want setenv, cd etc to modify the state of the parent. If they are executed by the child, the changes will go away when the child exits. For this built it functions, call the function inside execute instead of forking a new process.

Implementing Wildcards in Shell

No shell is complete without wildcards. Wildcards is a shell feature that allows one single command to be performed on multiple files that match the wildcard.

A wildcard describes filenames that match the wildcard. A wildcard works by iterating over all the files in the current directory or the directory described in the wildcard and then as arguments to the command those filenames that match the wildcard.

In general the “*” character matches 0 or more characters of any type. The character ”?” matches one character of any type.

To implement a wildcard, you should first translate the wildcard to a regular expression that a regular expression library can evaluate.

We suggest to implement first the simple case where you expand wildcards in the current directory. In shell.y, where arguments are inserted in the table do the expansion.

shell.y:

Before:

```
argument: WORD {
   Command::_currentSimpleCommand->insertArgument($1);
} ;
```

After:

```
argument: WORD {
  expandWildcardsIfNecessary($1);
} ;
```

The function expandWildcardsIfNecessary() is given next. Lines 4 to 7 will insert the argument the argument arg does not have "*" or "?" and return immediately. However, if these characters exist, then it will translate the wildcard to a regular expression.

```
 1  void expandWildcardsIfNecessary(char * arg)
 2  {
 3    // Return if arg does not contain "*" or "?"
 4    if (arg has neither "*" nor "?" (use strchr) ) {
 5          Command::_currentSimpleCommand->insertArgument(arg);
 6          return;
 7    }
 8
 9    // 1. Convert wildcard to regular expression
10    // Convert "*" -> ".*"
11    // "?" -> "."
12    // "." -> "\." and others you need
13    // Also add ^ at the beginning and $ at the end to match
14    // the beginning and the end of the word.
15    // Allocate enough space for regular expression
16    char * reg = (char*)malloc(2*strlen(arg)+10);
17    char * a = arg;
18    char * r = reg;
19    *r = '^'; r++; // match beginning of line
20    while (*a) {
21          if (*a == '*') { *r='.'; r++; *r='*'; r++; }
22          else if (*a == '?') { *r='.' r++;}
23          else if (*a == '.') { *r='\\'; r++; *r='.'; r++;}
24          else { *r=*a; r++;}
25          a++;
26    }
27    *r='$'; r++; *r=0;// match end of line and add null char
28    // 2. compile regular expression. See lab3-src/regular.cc
29    char * expbuf = regcomp( reg, â€¦ );
30    if (expbuf==NULL) {
31          perror("regcomp");
32          return;
```

```
33      }
34      // 3. List directory and add as arguments the entries
35      // that match the regular expression
36      DIR * dir = opendir(".");
37      if (dir == NULL) {
38            perror("oopendir");
39            return;
40      }
41      struct dirent * ent;
42      while ( (ent = readdir(dir))!= NULL) {
43            // Check if name matches
44            if (regexec(ent->d_name, re ) ==0 ) {
45            // Add argument
46            Command::_currentSimpleCommand->
47            insertArgument(strdup(ent->d_name));
48            }
49      }
50      closedir(dir);
51  }
52
```

The basic translations to be done from a wildcard to a regular expression are in the following table.

Wildcard Character	***Regular Expression***
"*"	".*"
"?"	"."
"."	"\."
Beginning of wildcard	"^"
End of Wildcard	"$"

In line 16 enough memory is allocated for the regular expression. Line 19. Insert the "^" to match the beginning of the regular expression with the beginning of the filename since we want to force a match of the whole filename. Line 20 to 26 convert the wildcard characters in the table above to the corresponding equivalents of the regular expression. Line 27 adds the "$" that matches the end of the regular expression with the end of the file name.

Lines 29 to 33 compile the regular expression into a more efficient representation that can be evaluated and it stores it in *expbuf*. Line 41 opens the current directory and lines 42 to 48

iterates over all the file names in the current directory. Line 44 checks if the filename matches the regular expression and if it is true then a copy of the filename will be added to the list of arguments. All this will add the file names that match the regular expression to the list of arguments.

Sorting Directory Entries

Shells like bash sort the entries matched by a wildcard. For example “echo *” will list all the entries in the current directory sorted. To have the same behavior, you will have to modify the wildcard matching as follows:

Line 5 creates a temporal array that will hold the file names matched by the wildcard. The initial size of the array is maxentries=20. The while loop in line 7 iterates over all the directory entries. If they match it will insert them into the temporal array . Line 10 to 14 will double the size of the array if the number of entries has reached the maximum limit. Line 20 will sort the entries using the sorting function of your choice. Finally, lines 23 to 26 iterate over the sorted entries in the array and add them as argument in sorted order.

```
 1
 2  struct dirent * ent;
 3  int maxEntries = 20;
 4  int nEntries = 0;
 5  char ** array = (char**) malloc(maxEntries*sizeof(char*));
 6
 7  while ( (ent = readdir(dir))!= NULL) {
 8    // Check if name matches
 9    if (regexec(ent->d_name, expbuf) ) {
10          if (nEntries == maxEntries) {
11             maxEntries *=2;
12             array = realloc(array, maxEntries*sizeof(char*));
13             assert(array!=NULL);
14          }
15          array[nEntries]= strdup(ent->d_name);
16          nEntries++;
17    }
18  }
19  closedir(dir);
20  sortArrayStrings(array, nEntries); // Use any sorting function
21
22  // Add arguments
23  for (int i = 0; i < nEntries; i++) {
24    Command::_currentSimpleCommand->
25    insertArgument(array[i]));
26  }
27
28  free(array);
```

Wildcards and Hidden Files

Another feature of shells like bash is that wildcards by default will not match hidden files that start with the character ".". In UNIX hidden files start with "." like .login, .bashrc etc.
Files that start with "." should not be matched with a wildcard. For example "echo *" will not display "." and "..".

To do this, the shell will add a filename that starts with "." only if the wildcard also has a "." at the beginning of the wildcard. To do this, the match if statement has to be modified in the following way:. If the filename matches the wildcard, then only if the filename starts with '.' and the wildcard starts with '.' then add the filename as argument. Otherwise, if the file name does not start with "." then add it to the list of arguments.

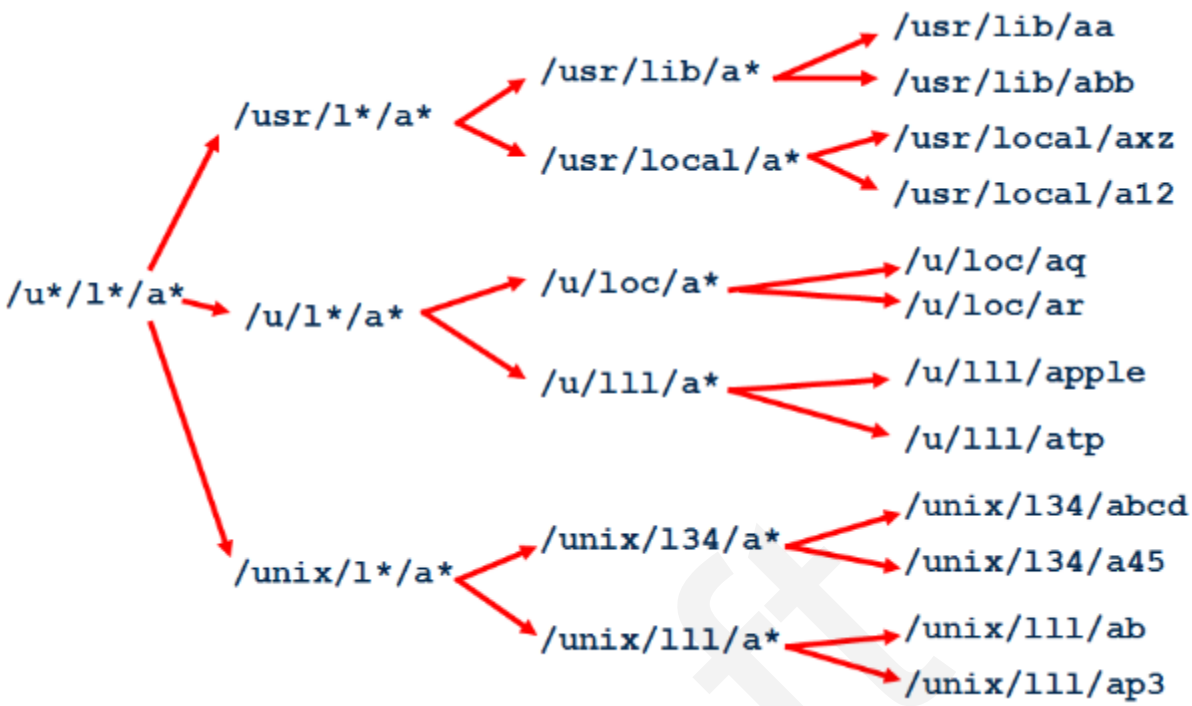
```
if (regexec (…) ) {
  if (ent->d_name[0] == '.') {
    if (arg[0] == '.')
      add filename to arguments;
    }
   }
   else {
     add ent->d_name to arguments
   }
}
```

Subdirectory Wildcards

Wildcards also may match directories inside a path:

For example, "echo /p*/*a/b*/aa*" will match not only the file names but also the subdirectories in the path.

To match subdirectories you need to match component by component



You may implement the wildcard Strategy in the following way.

Write a function expandWildcard(prefix, suffix) where

prefix- The path that has been expanded already. It should not have wildcards.

suffix – The remaining part of the path that has not been expanded yet. It may have wildcards.

The prefix will be inserted as an argument when the suffix is empty

expandWildcard(prefix, suffix) is initially invoked with an empty prefix and the wildcard in suffix. expandWildcard will be called recursively for the elements that match in the path.