**Indian Institute of Technology(ISM), Dhanbad**

**Department of Computer Science and Engineering**

**SHELL PROGRAMMING APPLICATION**

**Operating System Lab**

**OBJECTIVE:**

Implementation of grammar in shell.l and shell.y to make our parser interpret the command lines andprovide our executor with the correct information and performs below task.

**PROJECT DESCRIPTION:**

The shell is a program that interacts with the user through a terminal or takes the input from a fileand executes a sequence of commands that are passed to the Operating System.

**SOLUTION:**

The shell implementation is divided into three parts:

* Parser
* Executor
* Shell Subsystems

**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 beexecuted.

#define SHa\_RL\_BUFFSIZE 1024 //initial size of the buffer used to read line(RL)

/\*

Function Description : char \* SHa\_READLINE();

@brief Read a line of input from stdin.

@return The line from stdin.

\*/

char \* SHa\_READLINE(void){

intbuffsize = SHa\_RL\_BUFFSIZE;

int position = 0; // variable used as a pointer for the buffer array

//allocating the buffer

char \* buffer = malloc(sizeof(char) \* buffsize);

intch;

//variable for inputting characters from stdin

//Buffer Allocation Error

if(!buffer){

fprintf(stderr, "SHa: RL\_Buffer Allocation Error\n");

exit(EXIT\_FAILURE);

}

//inputting the characters

while(1){

//Read a character

ch = getchar();

//checking if we have reached EOF or not

if (ch == EOF) {

exit(EXIT\_SUCCESS);

}

else if(ch == '\n'){

//endline character is seen as end of the line

buffer[position] = '\0';

return buffer;

}

else{

buffer[position] = ch;

}

position++;

//If we have exceeded the buffer, reallocate the buffer.

if(position >= buffsize){

buffsize += SHa\_RL\_BUFFSIZE;

buffer = realloc(buffer, buffsize);

//Buffer Allocation Error

if(!buffer) {

fprintf(stderr, "SHa: RL\_Buffer Allocation Error\n");

exit(EXIT\_FAILURE);

}

}

}

}

To parse the line into a list of arguments. We are going to make a glaring simplification here, and say that we won’t allow quoting or backslash escaping in our command line arguments. Instead, we will simply use whitespace to separate arguments from each other. So the command echo "this message" would not call echo with a single argument this message, but rather it would call echo with two arguments: "this and message".

With those simplifications, all we need to do is “tokenize” the string using whitespace as delimiters. That means we can break out the classic library function “strtok”.

#define SHa\_TOK\_BUFFSIZE 64 //Initial size of the buffer where we split the input line and store the commands (tokens) to be passed to the executioner

#define SHa\_TOK\_DELIM" \t\r\n\a" //Delimiters according to which input string is to splitted

/\*

Function Description : char \*\* SHa\_SPLITLINE(char \* );

@brief Split a line into tokens (very naively).

@param line The line.

@return Null-terminated array of tokens.

\*/

char \*\*SHa\_SPLITLINE(char \* input){

intbuffsize = SHa\_TOK\_BUFFSIZE;

int position = 0; //as a pointer to buffer

//allocating tokens buffer

char \*\* tokens = malloc(buffsize \* sizeof(char\*));

char \*token, \*\*tokens\_backup;

//Tokens Buffer Allocation Error

if(!tokens){

fprintf(stderr, "SHa: TOK\_Buffer Allocation Error\n");

exit(EXIT\_FAILURE);

}

//splitting the input string

token = strtok(input , SHa\_TOK\_DELIM);

//storing splitted words in tokens buffer

while(token != NULL){

tokens[position] = token;

position++;

//If we have exceeded the buffer, reallocate the buffer.

if(position >= buffsize){

buffsize += SHa\_TOK\_BUFFSIZE;

tokens\_backup = tokens;

tokens = realloc(tokens, buffsize \* sizeof(char\*));

//Tokens Buffer Allocation Error

if(!tokens) {

free(tokens\_backup);

fprintf(stderr, "SHa: TOK\_Buffer Allocation Error\n");

exit(EXIT\_FAILURE);

}

}

token = strtok(NULL , SHa\_TOK\_DELIM);

}

tokens[position] = NULL;

return tokens;

}

At the start of the function, we begin tokenizing by calling strtok. It returns a pointer to the first token. What strtok() actually does is return pointers to within the string you give it, and place \0 bytes at the end of each token. We store each pointer in an array (buffer) of character pointers.

Finally, we reallocate the array of pointers if necessary. The process repeats until no token is returned by strtok, at which point we null-terminate the list of tokens.

**EXECUTOR**

The executor takes the command table generated by the parser and for every

SimpleCommand in the array it creates a new process. It also if necessary creates pipes

to communicate the output of one process to the input of the next one. Additionally, it

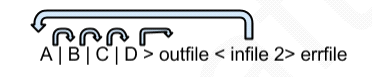
redirects the standard input, standard output, and standard error if there are any redirections.

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.



In the child process, we want to run the command given by the user. So, we use one of the many variants of the exec system call, execvp. The different variants of exec do slightly different things. Some take a variable number of string arguments. Others take a list of strings. Still others let us specify the environment that the process runs with. Our executioner expects a program name and an array (also called a vector, hence the ‘v’) of string arguments (the first one has to be the program name). The ‘p’ means that instead of providing the full file path of the program to run, we are going to give its name, and let the operating system search for the program in the path.

If the exec command returns -1 (or actually, if it returns at all), we know there was an error. So, we use perror to print the system’s error message, along with our program name, so users know where the error came from. Then, we exit so that the shell can keep running.

/\*

FUNCTION 1 :SHa\_EXECUTE()

DESCRIPTION :

This is function of execution for the shell program.

This function is to execute the shell(built-in or launch the shell program).

args is NUll terminated list of arguments passed to the function.

It returns 1 if the shell has to continue reading .

It returns 0 if termination is required .

\*/

intSHa\_EXECUTE(char \*\*args)

{

if (args[0] == NULL)

{

return 1;

}

inti;

for (i = 0; i<SHa\_num\_builtins(); i++)

{

if (strcmp(args[0], builtin\_str[i])==0)

{

return (\*builtin\_func[i])(args);

}

}

returnSHa\_LAUNCH(args);

}

/\*

FUNCTION 2: SHa\_LAUNCH()

DESCRIPTION :

This is function to launch the shell program and then wait for it to terminate.

args is NUll terminated list of arguments passed to the function.

It always returns 1 so that the execution can continue .

\*/

intSHa\_LAUNCH(char \*\*args)

{

pid\_tpid;

int status;

pid = fork();

if (pid == 0)

{

if (execvp(args[0], args) == -1)

{

perror("SHa");

}

exit(EXIT\_FAILURE);

}

else if (pid< 0)

{

perror("SHa");

}

else

{

do

{

waitpid(pid, &status, WUNTRACED);

} while (!WIFEXITED(status) && !WIFSIGNALED(status));

}

return 1;

}

/\*

FUNCTION 3: SHa\_LOOP()

DESCRIPTION :

This is function to take input from the parser and then execute it.

It does not return anything and only performs the execution .

\*/

voidSHa\_LOOP(void)

{

char \*line;

char \*\*args;

int status;

do

{

printf("> ");

line = SHa\_READLINE();

args = SHa\_SPLITLINE(line);

status = SHa\_EXECUTE(args);

free(line);

free(args);

}

while (status);

}

**Shell Subsystem:**

Other subsystems that complete our shell are:

● **Environment Variables:** Expressions of the form ${VAR} are expanded with thecorresponding environment variable.Also the shell should be able to set, expand andprint environment vars.

● **Wildcards:** Arguments of the form a\*a are expanded to all the files that match them inthe local directory and in multiple directories.

● **Subshells:** Arguments between `` (backticks) are executed and the output is sent asinput to the shell.

***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 allthe 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 aregular 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.

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

voidexpandWildcardsIfNecessary( char \* arg)

{

if( arg has neither “ \* ” nor “ ? ” ( usestrchr) ) {

Command :: \_currentSimpleCommand>

insertArgument( arg);

return;

}

char\* reg= ( char \*) malloc( 2 \* strlen( arg)+ 10 );

char\* a = arg;

char\* r = reg;

\* r = ‘ ^’ ; r++; // match beginning of line

while(\* a ) {

if(\* a == ‘ \* ’ ) { \* r = ’ . ’ ; r++; \* r = ’\* ’ ; r++; }

else if (\* a == ‘ ? ’ ) { \* r = ’ . ’ r ++;}

else if (\* a == ‘ . ’ ) { \* r = ’\\’ ; r ++; \* r = ’. ’ ; r++;}

else{ \* r =\* a ; r ++;}

a++;

}

\* r = ’$’ ; r++; \* r = 0 ;

char\* expbuf= regcomp( reg, â€¦ );

if( expbuf== NULL ) {

perror( “regcomp” );

return;

DIR \* dir= opendir(“. ”);

if( dir== NULL ) {

perror( “oopendir” );

return;

}

structdirent\* ent;

while( ( ent= readdir( dir))!= NULL ) {

if( regexec( ent>d\_name, re ) == 0 ) {

Command :: \_currentSimpleCommand>

insertArgument( strdup( ent>d\_name));

}

}

closedir( dir);

}

**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 we need to match component by component.