# Minishell

Tutorials

## What is Shell?

> 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.

- sh : Shell program. The original shell program in UNIX.
- bash : The GNU shell. Takes the best of all shell programs. It is currently the most common shell program.

### Parts of a Shell Program

1. **The Parser**

   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.

   ▼ Using Lex and Yacc to implement the Parser

   These tools are used to implement compilers, interpreters, and preprocessors.

   **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.

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

   ***Fig 4: Shell Grammar in BackusNaur Form***

   ▼ 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.

   ***Command :*** **ls -al | grep me > file1**

   **IO Redirection**

| | |
|---|---|
| **in** | Default |
| **out** | file1 |
| **err** | Default |

   **SimpleCommand array**

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

   ▼ Represent the command table

   **Command Data Structure**

```
// Describes a simple command and arguments

struct SimpleCommand
{
  int _numberOfAvailableArguments;
  int _numberOfArguments;
  char ** _arguments;

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

// 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;
};
```

▼ 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.

2. **The Executor**

   It will take the command table generated by the parser and for every SimpleCommand in the array it will create a new process.

   Create pipes to communicate the output of one process to the input of the next one.

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

3. **Shell Subsystems**

   a. **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.

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

   c. **Subshells**

      Arguments between `` (backticks) are executed and the output is sent as input to the shell.

## Bash - Shell Syntax

## Tutorial

## Brief Start

▼ Get/Redirect the unix signals

1. ctrl-C displays a new prompt on a new line.
2. ctrl-D exits the shell.
3. ctrl-\ does nothing.

```
// ignore Ctrl-\ Ctrl-C Ctrl-Z signals
ignore_signal_for_shell();
```

```
// user pressed ctrl-D
if (feof(stdin)) {
  exit(0);
  return 0;
}
```

▼ Check the grammar and syntax and either produce an error or an output the resulting command.

```
// keep getline in a loop in case interruption occurs
int again = 1;
while (again) {
  again = 0;
  printf("%s", getprompt());
  linebuffer = NULL;
  len = 0;
  ssize_t nread = getline(&linebuffer, &len, stdin);
  if (nread <= 0 && errno == EINTR) {
    again = 1;          // signal interruption, read again
    clearerr(stdin);  // clear the error
  }
}
```

▼ Lexically analyze and build a list of tokens

`lexer_build(linebuffer, len, &lexerbuf);`

▼ Try and run the array of tokens through a parser which parses them into an Abstract Syntax Tree.

```
// parse the tokens into an abstract syntax tree
if (!lexerbuf.ntoks || parse(&lexerbuf, &exectree) != 0)
{
  continue;
}
execute_syntax_tree(exectree);
```

## The Syntax Tree Parser

After we get the tokens from the lexer, we feed them to the parser or the syntax tree builder.

For those who do not know what a syntax tree is, it is the most important part of the compiler/interpreter. Put simply, it is a binary tree-like data structure that holds tokens and operations in order of the execution. Now, there are automatic ways to actually define the whole functions that parses texts in syntax tree.

The shell language grammar is defined as follows in Backus–Naur form:

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<letter> ::= "A" | "B" | "C" | "D" | "E" | "F"
<number> ::= <digit> | <letter>
<integer> ::= <number> | <number><integer>
```

The purpose is to recursively check if the order of tokens belongs to a particular grammar. This reduces down to check for terminal symbols and non-terminals. In case of a non-terminal the function sets equivalent to that production is simply called

### ▼ Syntax Tree란?

참조 링크 : **Abstract Syntax Tree (AST) 란 무엇인가?**

Tree는 프로그래밍 언어로 쓰여진 소스코드의 abstract syntactic 구조를 표현하기 위해서 사용.

쉽게 말하자면 특정 프로그래밍 언어로 작성된 프로그램 소스 코드를 각각 의미별로 분리하여 컴퓨터가 이해할 수 있는 구조로 변경시킨 트리를 의미

"*Abstract Syntactic*"란 프로그래밍 언어의 문법 및 각 문단의 역할을 표현하기 위한 규칙인데, data의 구조와 종류만 포함. 즉, "statement", "expression", "identifier" 과 같이 나눌 수 있도록 정의하는 문법 표현을 위한 규칙 " 이라고 Abstract Syntactic 을 이해하면 되겠다.

## FUNCTIONS

### ▼ Functions

- **readline()**

  get a line from a use with editing.

  ```
  #include <stdio.h>
  #include <readline/readline.h>
  #include <readline/history.h>
  ```

  readline will read a line from the terminal and return it, using
  prompt as a prompt. The line returned is allocated with malloc(3);
  the caller must free it when finished.

  **RETURN VALUE**

  The text of the line read. A blank like returns the empty string. If EOF is encountered while reading a line, and the line is empty, NULL is returned.  If an EOF is read with
  a non-empty line, it is treated as a newline.

## LEXER

## *The Lexical Analyzer*

The Minishell project involves handling Unix signals, checking grammar and syntax, lexically analyzing and building a list of tokens, and parsing them into an Abstract Syntax Tree. The Lexical Analyzer breaks the input string into tokens, groups characters inside single or double quotations, and expands wildcard characters using the glob POSIX function.

Simply break the input string from the user to a series of tokens. Now tokens could either be a character or a series of characters. It also groups characters that are inside a single or double quotations.

It is also able to expand wildcard characters. This is done via simply using the glob POSIX function.

#### ▼ Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell.

1. / (Escape Character)

   따옴표로 묶이지 않은 /는 <newline>을 제외하고 다음 문자의 리터럴 값을 보존해야 합니다. 백슬래시 뒤에 <new line>이 있으면 셸은 이를 라인 연속으로 해석해야 합니다. 입력을 토큰으로 분할하기 전에 백슬래시와 <newline>을 제거해야 합니다. 이스케이프된 <newline>은 입력에서 완전히 제거되고 공백으로 대체되지 않으므로 토큰 구분자 역할을 할 수 없습니다.

2. ‘

   단일 따옴표( ")로 문자를 둘러싸면 단일 따옴표 내 각 문자의 리터럴 값이 보존됩니다. 단일 따옴표는 단일 따옴표 내에서 발생할 수 없습니다.

3. “

   이중 따옴표( " " )로 묶은 문자는 다음과 같이 $, ` 및 / 를 제외한 이중 따옴표 내의 모든 문자의 문자 값을 보존해야 합니다.

   a. $

      파라미터 확장, 명령 대체 및 산술 확장을 도입하는 특별한 의미를 유지해야 합니다. 따옴표로 묶인 문자열 내의 "$("와 일치하는 "")" 사이에 포함된 입력 문자는 이중 따옴표의 영향을 받지 않아야 하며, 단어가 확장될 때 출력이 "$(...)"를 대체하는 명령을 정의해야 합니다. 동봉된 "${"에서 일치하는 "}"까지의 문자 문자열 내에서 이스케이프되지 않은 이중 따옴표 또는 단일 따옴표(있는 경우)가 짝수로 발생합니다. 리터럴 '{' 또는 '}'을(를) 이스케이프하려면 이전 백슬래시 문자를 사용해야 합니다. 매개 변수 확장의 규칙을 사용하여 일치하는 '}'을(를) 결정해야 합니다.

   b. `

      다른 형태의 명령 대체를 도입하는 특별한 의미를 유지해야 합니다. 초기 백쿼트에서 따옴표로 묶인 문자열과 백쿼트 앞에 없는 다음 백쿼트까지의 문자 부분에서 이스케이프 문자가 제거된 경우 출력이 "`...`"를 대체하는 명령을 정의합니다.

   c. \

      특수한 것으로 간주될 때 다음 문자 중 하나가 이어지는 경우에만 이스케이프 문자(이스케이프 문자(백슬래시))로서 특별한 의미를 유지해야 합니다 : `$   `   "   \   <newline>`

## ▼ enum

```
enum days
{
  sun, //0
  mon, //1
  tue, //2
  wed, //3
  thur,//4
  fri, //5
  sat  //6
};
enum days today;
today = sun;
```

타입명 안에 있는 원소들만 넣을 수 있음. 새로운 값을 지정할 수 없음.

실제로는 숫자로 대입되지만, 보기 편하게 원소 이름으로 보여짐.

Enumerated, 나열된 데이터들에 일정한 값 부여

열거된 데이터에는 정수값을 대응

나열된 데이터 목록에 대해 0부터 1씩 증가된 값이 자동 할당

**특정한 값이 대입되어 있으면 그 값에서 1씩 증가**

```
enum fresh
{
  apple,      // 0
  pear,       // 1
  orange = 6, // 6
  lemon.      // 7
} myfruit;

enum fresh yourfruit;

yourfruit = apple;
myfruit = lemon;
```

### 열거형의 다른 예

- bool

  열거 Bool형 변수 bool은 False나 True만 대입할 수 있고 내부적으로 0이나 1의 값을 가짐

  ```
  enum Bool{False, True}bool;
  bool = False;
  ```

- Multiple variables

열거 color형 변수 c1, c2, c3는 green, blue, yellow, white만 대입할 수 있고 내부적으로 3, 4, 8, 9의 값을 가짐

```
enum color
{
  green = 3, // 3
  blue,      // 4
  yellow = 8,// 8
  white.     // 9
}c1, c2, c3;

c1 = blue; // c1 = 4;
```

열거형을 사용하면 정수형 수를 이용하여 대입하는 것보다 의미있는 단어를 직접 대입하므로 프로그램을 해석하기가 쉬워짐

**ex)**

```
#include <stdio.h>
int main(void)
{
  enum days
  {
    sun,
    mon,
    tue,
    wed,
    thur = 7,
    fri,
    sat
  }a1, a2, a6;
  enum days a8;

  a8 = sun;
  a1 = sun;
  a2 = mon;
  a6 = fri;
  printf("a1 = %d, a6 = %d, a8 = %d\n", a1, a6, a8);
  return (0);
}

// a1 = 0, a6 = 8, a8 = 0
```

## PARSER

### *The Parser*

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.

## EXECUTOR

Traverse through out the tokens in order and make executions if necessary.

### ▼ Basic loop of a shell

For the basic program logic : handling commands with 3 steps. Read/Parse/Execute.

```
void shell_loop(void)
{
  char *line;
  char **args;
  int status;

  do {
    printf("> ");
    line = read_line();
    args = split_line(line);
    status = executer(args);

    free(line);
    free(args);
  } while (status);
}

int main(int argc, char **argv)
{
  shell_loop();
  return EXIT_SUCCESS;
}
```

## ▼ How shells start processes

2 ways of starting processes on Unix

1. By being ***Init***. When a Unix computer boots, its kernel is loaded. Once it is loaded and initialized, the kernel starts only one process, which is called ***Init.*** This process runs for the *Entire length of time* computer is on, and it manages loading up the rest of the processes that that user need for the user's computer to be useful.
2. `fork()` *system call.* When this function is called, the operating system makes a duplicate of the process and starts them both running. `fork()` returns 0 to the child and it returns to the parent the PID of its child. This means that the only way for new processes is to start is by an existing one duplicating itself.

When the user want to run a new process, the user wants to run a different program. That's what the `exec()` system call is all about. It replaces the current running program with an entirely new one. When `exec()` is called, the operating system stops the process, loads up the new program, and starts that one in its place. A process never returns from an `exec()` call.

---

### How most programs are run on Unix

An existing process `forks` itself into two separate ones. Then the **child** uses `exec()` to replace itself with a new program. The **parent** process can continue doing other things, and it can even keep tabs on its children, using the system call `wait()`.

```
int launch(char **args)
{
  pid_t pid, wpid;
  int status;

  pid = fork();
  if (pid == 0)
  {
    // Child process
    if (execvp(args[0], args) == -1)
    {
      perror("lsh");
    }
    exit(EXIT_FAILURE);
  }
  else if (pid < 0)
  {
    // Error forking
    perror("lsh");
  }
  else
  {
    // Parent process
    do
    {
      wpid = waitpid(pid, &status, WUNTRACED);
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
  }

  return 1;
}
```

Once `fork()` returns, we actually have 2 processes running concurrently. The **child** will take the first if condition (pid == 0).

To run the given command in the child process, use one of the `exec` system call, `execvp`. This particular variant expects a program name and an array(also called a vector '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're 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), there was an error. So, use `perror()` to print the system's error message. So users know where the error came from. Then, we exit so that the shell can keep running.

'*pid < 0*' checks whether `fork()` had an error. If so, we print it and keep going - there's no handling that error beyond telling the user and letting them decide if they need to quite.

Else, `fork()` executed successfully. The **parent** will land here. **Parent** needs to wait for the command to finish running in the **child**. We use `waitpid()` to wait for thew process's state to change. Processes can change state in lots of ways and not all of them meanthat the process has ended. A process can either exit or it can be killed by a signal. So, we use the macros to wait until either the processes are exited or killed. Then, the function finally returns a 1, as a signal to the calling function that we should prompt for input again.

### ▼ Redirection

### ▼ Heredoc

```
int fd = open("",  O_TMPFILE | O_RDWR, S_IRUSR | S_IWUSR);
```

- **O_TMPFILE :** create an unnamed temporary file that doesn't have a visible name in the file system directory hierarchy. The file is not linked to any specific name, so it doesn't have a permanent presence on disk.
- **O_RDWR :** the file should be opened for both reading and writing.
- **S_IRUSR :** "Read permission for the owner"
- **S_IWUSR :** "Write permission for the owner"

## Tests

### Redirection of heredoc

```
< test.txt << stop
```

It doesn't return the history on the terminal and no changes in the test.txt file.

```
<< test.txt
```

It doesn't return the history on the terminal and no changes in the test.txt file.

```
<< stop > test.txt
```

It doesn't return the history on the terminal and DELETE all contents in the test.txt file.

```
<< stop1 << stop2 << stop3
```

To escape the heredoc, it needs to type the delimiters in an order.

ex 1) > stop1 > stop2 >stop3

ex 2) > stop3(it doesn't count) > stop1 > stop2 > stop3

### Delimiter in quotes

```
$ << stop
> hi
> stop
$
```

```
$ << "stop"
> hi
> "stop"
> 'stop'
> stop
$
```

```
$ << stop
> hi
> "stop"
> 'stop'
> stop
$
```

As setting delimiter, quotes are not included as parts of delimiter.

## ▼ glob_t

`glob_t` is a structure defined in the `<glob.h>` header file. It is used to store the results of path name matching performed by the `glob` function.

uses the `glob_t` structure to perform pathname expansion on a
given token and replaces it with the matched pathnames, creating new
tokens for each matched pathname.

▼ Memebers

- `size_t gl_pathc`: The number of matched pathnames.
- `char** gl_pathv`: A pointer to an array of strings that contain the matched pathnames.
- • `size_t gl_offs`: The number of reserved positions at the beginning of `gl_pathv` for implementation use (usually 0).

1. `glob_t globbuf;`: Declares a `glob_t` structure named `globbuf` to store the results of the pathname matching.
2. `glob(token->data, GLOB_TILDE, NULL, &globbuf);`: Executes the `glob` function to perform pathname expansion on the `token->data` string. The `GLOB_TILDE` flag is used to expand the tilde (~) character in the pathname. The results are stored in the `globbuf` structure.
3. `if (globbuf.gl_pathc > 0)`: Checks if any pathnames were matched.
4. Inside the if statement, the code handles the matched pathnames by iterating over `globbuf.gl_pathv` and performing the following steps:
   - Increases the counter `k` by the number of matched pathnames (`globbuf.gl_pathc`).
   - Saves the next token in the variable `saved` for later attachment.
   - Replaces the current token with the first matched pathname by dynamically allocating memory, copying the first pathname to `token->data`, and updating its type.
   - Iterates over the remaining matched pathnames (`globbuf.gl_pathv`) and creates new tokens for each of them, attaching them to the linked list of tokens.
   - Restores the original token sequence by setting `token->next` to the saved token.

## ▼ static variable

1. 프로그램이 종료될 때 까지 값을 유지 (프로그램이 시작할 때 부터, main함수 종료시 까지)
2. 처음 실행시 한번만 초기화되고 초기화 값이 없으면 0으로 초기화 됨
3. 스택(지역 변수)이 아닌 정적데이터 영역(bss(비초기화 데이터) / data(초기화 데이터))을 사용
4. local static 변수는 해당 블록 내에서만 접근 가능

```
#include <stdio.h>

void sub(void)
{
  int x = 10;
  static int y = 10; //처음 실행시 한번만 초기화
  printf("x = %d, y = %d\n", x, y);
  x++;
  y++;
}

int main(void)
{
  sub();
  sub();
  sub();
  return (0);
}

  //x = 10, y = 10
  //x = 10, y = 11
  //x = 10, y = 12
```

x는 매번 프로그램이 실행 될 때 새로 만들어지지만, y는 초기값을 유지한다.

## EXPANDER

Handle the special variable $?, which holds the exit status of the most recently executed command or pipeline that ran in the foreground. In practical terms, if you run a series of commands or a pipeline in your shell, the value of $? will be updated to reflect the exit status of the last command or pipeline executed. This information can be useful for scripting, error handling, or conditional execution of subsequent commands based on the success or failure of the previous ones.

### ▼ $?

$? is a special variable in Unix-like operating systems (e.g., Linux, macOS) that represents the exit status of the last executed command or pipeline in the shell. In other words, it holds the result of the most recent command's execution.

### ▼ Exit status

“Exit status” is a numeric value that indicates the outcome of a command's execution. By convention, a 0 exit status means success, and any non-zero value indicates an error or some other type of failure.

### ▼ Foreground pipeline

"Foreground pipeline" refers to a sequence of one or more commands that are executed in the foreground (as opposed to the background) in the shell. A pipeline typically consists of multiple commands connected by the pipe symbol "|", where the output of one command becomes the input for the next.

▼ Examples

```
$ ls
file1.txt file2.txt file3.txt
$ echo $?
0
```

In this example, the "ls" command lists the files in the current directory, and it executes successfully (exit status 0). After running the "ls" command, we immediately check the value of "$?" using "echo $?", and it returns 0.

```
$ mkdir new_directory
$ if [ $? -eq 0 ]; then echo "Directory created successfully!"; else echo "Error creating directory!"; fi
Directory created successfully!We create a new directory named "new_directory" using the "mkdir" command. After executing the "mkd
```

We create a new directory named "new_directory" using the "mkdir" command. After executing the "mkdir" command, we use "$?" in the if-else conditional statement to check its exit status. If the exit status is 0 (indicating success), the message "Directory created successfully!" is displayed. Otherwise, if there was an error during the directory creation, the message "Error creating directory!" is displayed.

## BUILTINS

Some of commands are built right into the shell. The reason is If you want to change directory, you need to use the function `chdir()`. The thing is, the current directory is a property of a process. So, if you wrote a program called `cd` that changed directory, it would just change its own current directory, and then terminate. Its parent process's current directory would be unchanged. Instead, the shell process itself needs to execute `chdir()`, so that its own current directory is updated. Then, when it launches child processes, they will inherit that directory too.

Also most shells are configured by running configuration scripts, like `~/.bashrc`. Those scripts use commands that change the operation of the shell. These commands could only change the shell's operation if they were implemented within the shell process itself.

S, it makes snes that we need to add some commands to the shell itself.

### ▼ Putting together builtins and processes

Implementing `executer()`, the function that will either launch a builtin, or a process.

```
int executer(char **args)
{
  int i;

  if (arg[0] == NULL)
  {
    //An empty command was entered.
    return (1);
  }
  for (i = 0, o < num_builtins(); i++)
  {
    if (strcmp(args[0], builtin_str[i]) == 0)
    {
      return (*builtin_func[i](args));
    }
  }
  return (launch(args));
}
```

All this does is check if the command equals each builtin, and if so, run it. If it doesn't match a builtin, it calls `launch()` the process. Then cone caveat is that `args` might just contain `NULL`, if the user entered an empty string, or just white space.