# Розділ 5. Написання власної оболонки

*Ви дійсно щось розумієте, поки не запрограмуєте це.*
*GRR*

## Вступ

Останній розділ розповідав про те, як використовувати програму оболонки за допомогою команд UNIX. Оболонка — це програма, яка взаємодіє з користувачем через термінал або отримує вхідні дані з файлу та виконує послідовність команд, які передаються до операційної системи. У цьому розділі ви дізнаєтеся, як написати власну програму оболонки.

#### Програми оболонки

Програма оболонка – це програма, яка дозволяє взаємодіяти з комп'ютером. У оболонці користувач може запускати програми, а також перенаправляти вхідні дані, щоб вони надходили з файлу, а вихідні дані — у файл. Оболонки також забезпечують конструкції програмування, такі як if, for, while, функції, змінні тощо. Крім того, програми оболонки пропонують такі функції, як редагування рядків, історія, завершення файлів, символи підстановки, розширення змінних середовища та конструкції програмування. Ось список найпопулярніших програм оболонки в UNIX:

| | |
|---|---|
| ш | Програма Shell. Оригінальна програма оболонки в UNIX. |
| csh | C Оболонка. Покращена версія sh. |
| tcsh | Версія Csh з редагуванням рядків. |
| кш | Корн Шелл. Батько всіх передових снарядів. |
| bash | Оболонка GNU. Бере найкраще з усіх програм оболонки. На даний момент це найпоширеніша програма оболонки. |

На додаток до оболонок командного рядка існують також графічні оболонки, такі як Windows Desktop, MacOS Finder або Linux Gnome і KDE, які спрощують використання комп'ютерів для більшості користувачів. Однак ці графічні оболонки не замінюють оболонки командного рядка для досвідчених користувачів, які хочуть виконувати складні послідовності команд багаторазово або з параметрами, недоступними в зручних, але обмежених графічних діалогових вікнах та елементах керування.

### Частини програми Shell

Реалізація оболонки розділена на три частини: **Парсер** , **Виконавець**, і**С пекло Підсистеми.**

## Парсер

Парсер — це програмний компонент, який читає командний рядок, наприклад «ls al», і поміщає його в структуру даних, що називається ***Таблиця команд***, в якому зберігатимуться команди, які будуть виконано.

## Виконавець

Виконавець візьме таблицю команд, згенеровану синтаксичним аналізатором, і для кожної SimpleCommand в масиві створить новий процес. Він також, якщо необхідно, створить канали для передачі результатів одного процесу на вхід наступного. Крім того, він переспрямує стандартний вхід, стандартний вихід і стандартну помилку, якщо є якісь переспрямування.

На малюнку нижче показано командний рядок «A | B | C | D". Якщо є переспрямування, наприклад « **< *infile*** " виявлене синтаксичним аналізатором, вхід першої SimpleCommand A перенаправляється з ***infile***. Якщо є перенаправлення виводу, наприклад « **> *outfile*** " , він перенаправляє вихідні дані останнього SimpleCommand (D) для ***outfile*** .


Якщо є перенаправлення до errfile, наприклад « **> & файл помилок**" stderr усіх SimpleCommand процеси будуть перенаправлені на **errfile**

#### Підсистеми оболонки

Інші підсистеми, які завершують вашу оболонку:

- Змінні середовища: вирази форми ${VAR} розширюються відповідною змінною середовища. Також оболонка повинна мати можливість встановлювати, розширювати та друкувати варі середовища.
- Підстановкові знаки: аргументи у формі a*a розгортаються до всіх файлів, які відповідають їм у локальному каталозі та в кількох каталогах.
- Підоболонки: Аргументи між `` (обратними галочками) виконуються, а вихідні дані надсилаються як вхідні дані до оболонки.

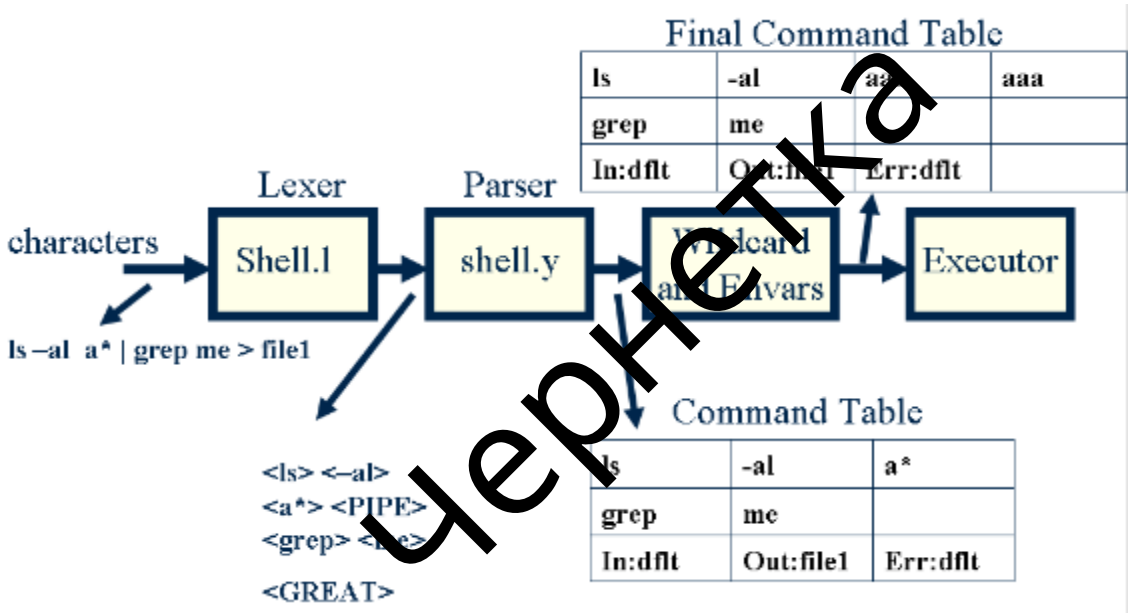
Ми настійно рекомендуємо вам реалізувати власну оболонку, дотримуючись кроків у цьому розділі. Реалізація вашої власної оболонки дасть вам дуже добре розуміння того, як взаємодіють програми інтерпретатора оболонки та операційна система. Крім того, це буде гарний проект, який можна показати під час співбесіди майбутнім роботодавцям.

**Використання Lex і Yacc для реалізації аналізатора**

Ви будете використовувати два інструменти UNIX для реалізації вашого аналізатора: Lex і Yacc. Ці інструменти використовуються для реалізації компіляторів, інтерпретаторів і препроцесорів. Вам не потрібно знати теорію компілятора, щоб використовувати ці інструменти. Усе, що вам потрібно знати про ці інструменти, буде пояснено в цьому розділі.

Парсер ділиться на дві частини: а**Л Ексичний аналізатор**або**ex**Lexer приймає вхідні символи і об'єднує героїв у слова, які називаються **токени**, і а**п дупка** що обробляє лексеми відповідно до граматики та побудувати таблицю команд.

Ось діаграма оболонки з Lexer, Parser та іншими компонентами.






***токени***описані у файлі**с** **пекло**використання регулярних виразів. Файл**shell.l** є обробляється програмою під назвою**лекс**який створює лексичний аналізатор.

Граматичні правила, які використовує синтаксичний аналізатор, описані у файлі з назвою**с** **пекло** використання синтаксису вирази, які ми описуємо нижче. **shell.y** обробляється програмою під назвою**у** **згідно**з що генерує програму аналізатора. I lex, і yacc є стандартними командами в UNIX. Ці команди можна використовувати для реалізації дуже складних компіляторів. Для оболонки ми будемо використовувати підмножину Lex і Yacc, щоб створити таблицю команд, необхідну для оболонки.

Вам потрібно реалізувати наведену нижче граматику***с пекло***і ***shell.y*** щоб наш парсер інтерпретував командних рядків і надати нашому виконавцю правильну інформацію.

```
cmd [arg]* [ | cmd [arg]* ]*
        [ [> ім'я файлу] [< ім'я файлу] [ >& ім'я файлу] [>> ім'я файлу] [>>& ім'я файлу] ]* [&]
```

***Рис. 4: Граматика оболонки у формі BackusNaur***

Ця граматика написана у форматі "*Форма BackusNaur*». Наприклад **cmd [arg]*** означає команда, ***cmd***, після чого 0 або більше аргументів, ***arg***. Вираз **[ | cmd [arg]* ]*** представляет додаткові підкоманди каналу, де їх може бути 0 або більше. Вираз **[ > ім'я файлу]** означає, що може бути 0 або 1 **> ім'я файлу** перенаправлення. The ***[&]*** в кінці означає, що ***&*** символ необов'язковий.

Приклади команд, прийнятих у цій граматиці:

**ls –al**

**ls –al > out**

**ls –al | відсортувати >& розібрати**

**awk –f x.awk | сортувати –u < infile > outfile &**



## Таблиця команд

***Таблиця команд*** масивом ***Проста команда*** структури. ***Проста команда*** *структурувати* містить члени для команди та аргументи ємого запису в конвеєрі. Синтаксичний аналізатор також перегляне к ий рядо визначить, чи є переспрямування введення або виведення на основі с волів, при ніх у к анді (тобто < infile або > outfile).

Ось приклад команди т ***Таблиця команд*** ін генерує:

**команда**

**ls al | grep me > file1**

***Таблиця команд***

**SimpleCommmand масив**

| | | | |
|---|---|---|---|
| **0:** | **ls** | **ін** | **НУЛЬ** |
| **1:** | **grep** | **мене** | **НУЛЬ** |

**Перенаправлення IO**:

| | | |
|---|---|---|
| **в: за замовчуванням** | **вихід: файл1** | **помилка: за замовчуванням** |

Для представлення таблиці команд ми будемо використовувати такі класи:**C наказ** і **SimpleCommand**.

```
//CommandDataStructure

//Describesasimplecommandandarguments
structSimpleCommand{
        //Availablespaceforargumentscurrentlypreallocated
        int_numberOfAvailableArguments;

        //Числофаргументи
        int_numberOfArguments;

        //Масиваргументи
        char**_аргументи;

        SimpleCommand();
        voidinsertArgument(char*argument);
};

//Описує повну команду з кількома фасонами труб //і
переспрямуванням введення/виведення.
structCommand{
        int_numberOfAvailableSimpleCommands;
        int_numberOfSimpleCommands;
        SimpleCommand**_simpleCommands;
        char*_outFile;
        char*_inputFile;
        char*_errFile;
        int_background;

        voidprompt();
        voidprint();
        voidexecute();
        voidclear();

        Команда ();
        voidinsertSimpleCommand(SimpleCommand*simpleCommand);

        staticCommand_currentCommand;
        staticSimpleCommand*_currentSimpleCommand;
};
```

Конструктор**C impleCommand::SimpleCommand** створює просту порожню команду. Метод **SimpleCommand::insertArgument( char * аргумент)** вставляє новий аргумент в SimpleCommand і, якщо необхідно, збільшує масив _arguments. Він також гарантує, що останній елемент має значення NULL, оскільки це потрібно для системного виклику exec().

Конструктор**C ommand::Command**( ) конструкції та порожня команда, яка буде заповнюється методом Command::**я nsertSimpleCommand(SimpleCommand* проста команда)..** insertSimpleCommand також збільшує масив _simpleCommands if необхідно. Змінні _outFile, _inputFile, _errFile будуть мати значення NULL, якщо переспрямування не було зроблено, або ім'я файлу, до якого вони переспрямовуються.

Змінні **_currentCommand** і_ **поточна команда** є статичними змінними, тобто є лише один на весь клас. Ці змінні використовуються для створення команд Command і Simple під час аналізу команди.

Класи Command і SimpleCommand реалізують основну структуру даних, яку ми будемо використовувати в оболонці.

### Реалізація лексичного аналізатора

Лексичний аналізатор розділяє введення на лексеми. Він зчитує символи один за одним зі стандартного введення. і сформуйте маркер, який буде переданий аналізатору. Лексичний аналізатор використовує файл ***c пекло*** який містить регулярні вирази, що описують кожен із токенів. Лексер буде читати введений символ за символом і намагатиметься узгодити введені дані з кожним із регулярних виразів. Коли рядок у вхідних даних відповідає одному з регулярних виразів, він виконає код {...} праворуч від регулярного виразу. Нижче наведено спрощену версію shell.l, яку використовуватиме ваша оболонка.

```
/*
 * shell.l:simplelexicalanalyzerfortheshell.
 */

%{

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

%}

%%

\n          {
                        повернутисяNEWLINE;
```

```
        }

[\t]    {
                /*Discardspacesandtabs*/
        }

">"     {
                return ВЕЛИКИЙ;
        }

"<"     {
                БЕЗ повернення;
        }

">>"      {
                returnGREATGREAT;
        }

">&"      {
                returnGREATAMPERSAND;
        }

“|”     {
                returnPIPE;
        }

«&»     {
                returnAMPERSAND;
        }


[ \t\n][ \t\n]* {
                /*Припустимо, що назви файлів мають тільки альфа-
                символи*/ yylval.string_val=strdup(yytext);
                returnWORD;
        }

/*Додати більше маркерів тут*/

.       {
                /*Недійсний вхід символу*/
                returnNOTOKEN;
        }

%%
```

Файл ***shell.l*** пропускається через ***lex*** т о створити файл C з назвою ***lex.yy.c.*** Цей файл реалізує сканер, який синтаксичний аналізатор використовуватиме для перекладу символів у токени.

Ось команда, яка використовується для запуску lex.

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

Файл ***lex.yy.c*** це файл C, який реалізує lexer для відокремлення лексем, описаних у ***shell.l***.

У shell.l є дві частини. Верхня частина виглядає так:

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

Це частина, яка буде вставлена у верхній частині файлу ***lex.yy.c*** безпосередньо без модифікації, яка включає файли заголовків і визначення змінних, які ви будете використовувати в сканері. Саме тут ви можете оголосити змінні, які використовуватимете у своєму лексері.

Друга частина відмежована ***%%*** виглядає так:

```
%%
\n      {
            повернути NEWLINE;
        }
[ \t]   {
            /* Відкинути пробіли та табуляції */
        }
">"     {
            повернути ВЕЛИКИЙ;
        }
[^ \t\n][^ \t\n]* {
            /* Припустимо, що імена файлів містять лише альфа-
            символи */ yylval.string_val = strdup(yytext);
            повернути СЛОВО;
        }
%%
```

Ця частина містить регулярні вирази, які визначають маркери, утворені шляхом взяття символів зі стандартного введення. Щойно токен буде сформований, він буде повернутий або в деяких випадках відкинутий. Кожне правило, яке визначає маркер, також має дві частини:

*регулярний вираз {*
*дії*
}

напр

\n {
повернути NEWLINE;
}

Перша частина — це регулярний вираз, який описує маркер, якому ми очікуємо відповідати. Дія — це фрагмент коду C, який додає програміст, який виконується, коли маркер відповідає регулярному виразу. У наведеному вище прикладі, коли знайдено символ нового рядка, lex поверне константу ***НОВИНКА*** . Пізніше ми опишемо, де ***НОВИНКА*** константи визначені.

Ось більш складна лексема, яка описує ***СЛОВО*** А ***ОРД*** може бути аргументом на користь а команду або саму команду.

```
[^\t\n][^\t\n]* {
                /* припустимо, що назви файлів мають тільки альфа-
                символи*/ yylval.string_val=strdup(yytext);
                return WORD;
}
```

Вираз у ***[...]*** відповідає будь-якому символу, що знаходиться в дужках. Вираз ***[^...]*** відповідає будь-якому символу, який не входить у дужки. тому ***[^\t\n][^ \t\n]**** описує а токен, який починається з символу, який не є пробілом, табуляцією або новим рядком, і за ним слідує нуль або більше символів, які не є пробілами, табуляціями чи новими рядками. Знайдений маркер знаходиться у змінній, що називається ***yytext*** . Після збігу слова призначається дублікат відповідного токена ***yylval.string_val*** в такій заяві:

yylval.string_val = strdup(yytext);

це спосіб, яким значення маркера передається аналізатору. Нарешті, константа ***ВОРД*** є повернуто до аналізатора.

**Додавання нових маркерів до shell.l**

***shell.l*** описаний вище наразі підтримує зменшену кількість маркерів. Як перший крок під час розробки вашої оболонки вам потрібно буде додати більше маркерів до нової граматики, яких зараз немає в***shell.l*** . Дивіться граматику в***малюнок 4***,щоб побачити, яких маркерів не вистачає і які потрібні бути доданим до***пекло*** Ось деякі з цих жетонів:

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

**Додавання нових маркерів до shell.y**

Ви додасте назви маркерів, які ви створили на попередньому кроці, до ***shell.y*** в маркері % розділ:

%токен NOTOKEN, GREAT, NEWLINE, WORD, GREATGREAT, PIPE, AMPERSAND тощо

## Завершення граматики

Вам потрібно додати більше правил до shell.y, щоб завершити граматику оболонки. На наступному малюнку синтаксис оболонки розділяється на різні частини, які будуть використовуватися для побудови граматики.



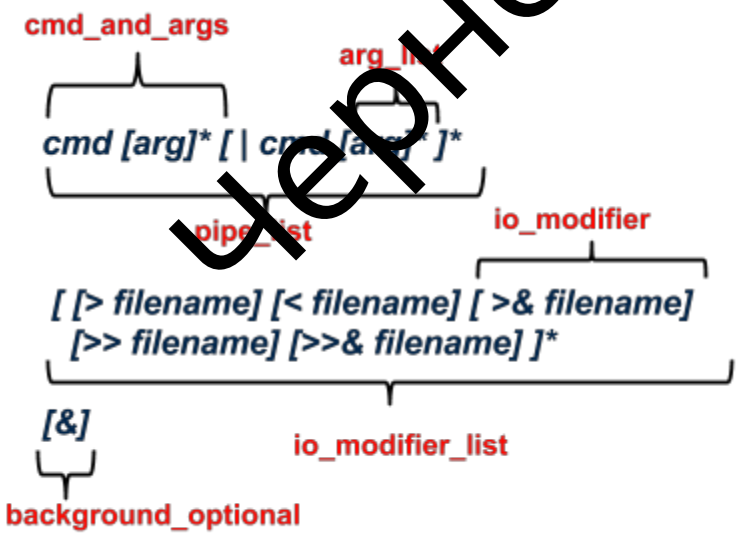



Figure 3. Shell Grammar labeld with the different parts.

Ось граматика, сформована за допомогою маркування, визначеного вище:

```
мета: список_команд;
```

```
arg_list:
        arg_list СЛОВО
        | /*пусто*/
                ;
cmd_and_args:
        WORD arg_list
                ;
pipe_list:
        pipe_list PIPE cmd_and_args |
        cmd_and_args
        ;

io_modifier:
        ВЕЛИКЕ ВЕЛИКЕ Слово
         | ВЕЛИКЕ СЛОВО
         | ВЕЛИКИЙ ВЕЛИКИЙ САМОМПЕРСАНД
         Слово | ВЕЛИКІ МІСЦІ І Слово
         | МЕНШЕ слова
        ;
io_modifier_list:
        io_modifier_list io_modifier
                | /*пусто*/
                ;
background_optional:
                Амперсанд
                | /*пусто*/
                ;
командний рядок:
         pipe_list io_modifier_list background_optional NEWLINE |
        NEWLINE /*прийняти порожній рядок замість*/
        | помилка NEWLINE{yyerrok;}
                /*відновлення помилки*/

список_команд:
                 command_list command_line ;/*
                командний цикл*/
```



Граматика вище реалізує цикл команд у самій граматиці.

Маркер помилки — це спеціальний маркер, який використовується для відновлення помилок. помилка аналізуватиме всі маркери, доки не буде знайдено відомий маркер, наприклад <NEWLINE>.*у* ***помилка*** повідомляє аналізатору, що була помилка видужав.

Синтаксичний аналізатор бере лексеми, згенеровані лексичним аналізатором, і перевіряє, чи відповідають вони синтаксису, описаному граматичними правилами в ***с пекло***. Під час перевірки, чи введено командний рядок дотримуючись синтаксису, синтаксичний аналізатор виконуватиме дії або частини коду C, які ви вставите між правилами граматики. Ці фрагменти коду називаються діями, і вони розмежовані фігурними дужками { action; }.

Вам потрібно додати дії {...} в граматику, щоб заповнити таблицю команд.

приклад:

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

**Створення процесів у вашій оболонці**

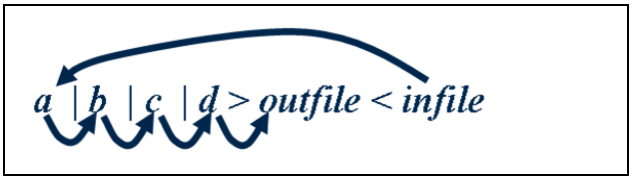
Почніть зі створення нового процесу для кожної команди в конвеєрі та змусити батьків чекати останньої команди. Це дозволить виконувати прості команди, такі як «cat».



```
Команда::виконати()
{
    інтрете;
    for(inti=0;i<_numberOfSimpleCommands;i++){
        ret=вилка();
        if(ret==0){
            //дитина
            execvp(sCom[i]>_args[0],sCom[i]>_args);
            perror(“execvp”);
            _exit(1);
        }
        elseif(ret<0){
            perror("вилка");
            повернення;
        }
        //Parentshellcontinue
    }//для
    if(!фон){
        // дочекатися останнього процесу
        waitpid(ret,NULL);
    }
}//виконати
```

**Перенаправлення каналу та введення/виводу у вашій оболонці**

Стратегія вашої оболонки полягає в тому, щоб батьківський процес виконував усі конвеєри та перенаправлення, перш ніж розділяти процеси. Таким чином діти успадкують перенаправлення. Батько має зберегти введення/виведення та відновити його в кінці. Stderr однаковий для всіх процесів




На цьому малюнку процес ***a*** надсилає вихід до ***cipe 1*** . Тоді ***б*** зчитує свій вхід з ***cipe 1*** і відправляє свій вихід на ***труба і2*** так далі. Остання команда ***d*** р бере свій вхід від ***труба і3*** відправити його вихід на *вихідний файл*. Вхідні дані надходять від ***infile***.

Наступний код показує, як реалізувати це переспрямування. Деякі перевірки помилок були виключені для простоти.

```
1 порожнеча Команда::виконати(){
2       //зберегти/вийти
3       міжнар tmpin =dropy(0);
4       міжнар tmpout =dropy(1);
5
6       //встановити початковий ввод
7       міжнар fdin ;
8       якщо( nfile ) {
9           fdin =оручка( nfile , "ЧИТАЙТЕ");
10      }
11      інше {
12          //Використовується введення за умовчанням
13          fdin =dropy( tmpin );
14      }
15
16      міжнар відкл;
17      міжнар fdout ;
18      для(я=0; я<numssimplecommands;я + +){
19          //redirectinput
20          dup2(fdin , 0);
21          закрити(fdin );
22          //налаштуванняввиведення
23          якщо( я= =п umssimplecommands 1) {
24              //Lastsimplecommand
25              якщо( outfile ) {
```

```
26              fdout =ручка(outfile     , â¦â€¦  ) ;
27            }
28            інше {
29                //Використовується вихід за умовчанням
30              fdout =dup(tmpout ) ;
31            }
32          }
33
34          інше {
35              //Неостанній
36              //проста команда
37              //createpipe
38              міжнаp fdpipe  [2] ;
39              труба(fdpipe   ) ;
40              fdout =fdpipe  [1] ;
41              fdin  =fdpipe  [0] ;
42          }//if/else
43
44          //Виведення перенаправлення
45          dup2(fдоут  , 1) ;
46          закрити(fдоут  ) ;
47
48          //Createchildprocess
49          відкеворк  ();
50          якщо(рет==0) {
51              execvp (cmd [я] .args  [0] ,c  cmd [я] .args  ) ;
52              помилка(â€œexecvpâ€  ) ;
53              _вихід(1) ;
54          }
55      }/ / для
56
57      //відновити/зняти за замовчуванням
58      dup2(tmpin  , 0) ;
59      dup2(tmpout , 1) ;
60      закрити(tmpin  ) ;
61      закрити(tmpout ) ;
62
63      якщо(!фоніпідґрунтя   ) {
64          //Зачекайте на останню команду
65          очікування(рет, H ULL  ) ;
66      }
67
68  }/ /виконати
```



Метод execute() є основою оболонки. Він виконує прості команди в окремому процесі для кожної команди і виконує перенаправлення.

Рядки 3 і 4 зберігають поточний stdin і stdout у два нових дескриптори файлу за допомогою функції dup(). Це дозволить наприкінці execute() відновити stdin і stdout так, як вони були

на початку виконання (). Причина цього полягає в тому, що stdin і stdout (дескриптори файлів 0 і 1) будуть змінені в батьківському під час виконання простих команд.

```
3       міжнар tmpin =dropy(0);
4       міжнар tmpout =dropy(1);
```

Рядки з 6 по 14 перевіряють, чи є вхідний файл перенаправлення в командній таблиці у формі “command < infile”. Якщо є переспрямування введення, файл відкриється в ***infile*** і збережіть його в ***fdin***. В іншому випадку, якщо немає переспрямування введення, він створить дескриптор файлу, який посилається на введення за замовчуванням. В кінці цього блоку інструкцій ***fdin*** буде дескриптором файлу, який має введення командного рядка, і його можна закрити, не впливаючи на програму батьківської оболонки.

```
6       //встановити початковий ввод
7       міжнар fdin ;
8       якщо( infile ) {
9           fdin =оручка(infile , ЧИТАЙТЕ);
10      }
11      інше {
12          //Використовується введення за умовчанням
13          fdin =dropy(tmpin );
14      }
```

Рядок 18 - це ***для***цикл, який повторює всі прості команди в таблиці команд. Це ***для***цикл створить процес для кожної простої команди і виконає з’єднання труб.

Рядок 20 перенаправляє стандартний вхід з ***fdin***. Після цього будь-яке читання з stdin буде походять із файлу, на який вказує ***fdin***. На першій ітерації введення першої простої команди прийде з ***fdin***. ***fdin*** буде перепризначено вхідній трубі пізніше в циклі. Лінія 21 закриється ***fdin*** оскільки дескриптор файлу більше не потрібен. Загалом, хороша практика закривати дескриптори файлів, якщо вони не потрібні, оскільки для кожного процесу доступно лише кілька (як правило, 256 за замовчуванням).

```
16      міжнар відкл;
17      міжнар fdout ;
18      для(я=0; я<numssimplecommands;я + +){
19          //redirectinput
20          dup2(fdin , 0);
21          закрити(fdin );
```

Рядок 23 перевіряє, чи відповідає ця ітерація останній простій команді. Якщо це так, він перевірить у рядку 25, чи є перенаправлення вихідного файлу у формі «команда > вихідний файл» і відкриється ***вихідний файл*** і призначити його ***fdout***. В іншому випадку в рядку 30 він створить новий дескриптор файлу вказує на вхід за замовчуванням. У цьому переконаються рядки з 23 по 32 ***f доут*** є файловим дескриптором для вихід на останній ітерації.

```
23          //налаштуваннявиведення
23          якщоя=  =п umssimplecommands  1){
24            //Lastsimplecommand
25            якщо outfile     ){
26              fdout =ручка(outfile     , â¦â€¦  );
27            }
28            інше {
29              //Використовується вихід за умовчанням
30              fdout =dropуприout );
31            }
32          }
33
34          інше {...
```

Рядки з 34 по 42 виконуються для простих команд, які не є останніми. Для цих простих команд виводом буде канал, а не файл. Рядки 38 і 39 створюють нову трубу. Нова труба. Канал — це пара дескрипторів файлів, які передаються через буфер. Все, що записано у файловому дескрипторе fdpipe[1], можна прочитати з fdpipe[0]. IN рядки 41 і 42 fdpipe[1] призначається fdout, а fdpipe[0] призначається fdin.

Рядок 4fdin dpipe 0]М може бути ядром реалізації труб, оскільки він робить введіть fdin наступної простої команди на наступній ітерації, щоб отримати від fdpipe[0] поточної простої команди.

```
34          інше {
35              //Неостанній
36              //проста команда
37              //createpipe
38              міжнаfdpipe  [2] ;
39              трубаfdpipe   ) ;
40              fdout =fdpipe   [1] ;
41              fdin  =fdpipe   [0] ;
42          }//if/else
43
```

Рядки 45 перенаправляють стандартний вивід для переходу до об’єкта файлу, на який вказує fdout. Після цього рядка stdin і stdout були перенаправлені або до файлу, або до каналу. Рядок 46 закриває fdout, який більше не потрібен.

```
44          //Виведення перенаправлення
45          dup2(fdout , 1);
46          закрити(fdout );
```

Коли програма оболонки знаходиться в рядку 48, переспрямування введення та виведення для поточної простої команди вже встановлено. Рядок 49 розгалужує новий дочірній процес, який успадковуватиме дескриптори файлу 0,1 і 2, які відповідають стандартним вихідним кодам, стандартним виводам і stderr, які перенаправляються або на термінал, і на файл, або на канал.

Якщо під час створення процесу немає помилки, рядок 51 викликає системний виклик execvp(), який завантажує виконуваний файл для цієї простої команди. Якщо execvp вдасться, він не повернеться. Це пов'язано з тим, що в поточному процесі було завантажено новий виконуваний образ, а пам'ять була перезаписана, тому немає до чого повертатися.

```
48          //Createchildprocess
49          відгалузити ();
50          якщо (ret==0) {
51              execvp (cmd [я] .args [0] ,c cmd [я] .args );
52              помилка(«execvp». );
53              _вихід(1);
54          }
55      }// / для
```

Рядок 55 – це кінець циклу for, який виконує ітерацію всіх простих команд.

Після виконання циклу for всі прості команди виконуються у власному процесі, і вони спілкуються за допомогою каналів. Оскільки stdin і stdout батьківського процесу були змінені під час переспрямування, рядки 58 і 59 викликають dup2, щоб відновити stdin і stdout до того самого об'єкта файлу, який був збережений у tmpin і tmpout. В іншому випадку оболонка отримає вхідні дані з останнього файлу, до якого було перенаправлено введення. Нарешті, рядки 60 і 61 закривають тимчасові дескриптори файлів, які були використані для збереження stdin і stdout процесу батьківської оболонки.

```
57          //відновити/зняти за замовчуванням
58          dup2(tmpin , 0);
59          dup2(tmpout , 1);
60          закрити(tmpin );
61          закрити(tmpout );
```

Якщо фоновий символ «&» не встановлено в командному рядку, це означає, що батьківський процес оболонки повинен зачекати завершення останнього дочірнього процесу в команді, перш ніж друкувати підказку оболонки. Якщо фоновий символ «&» встановлено, це означає, що командний рядок буде

запускати асинхронно з оболонкою, тому процес батьківської оболонки не чекатиме завершення команди й негайно надрукує підказку. Після цього виконання команди виконано.

```
63          якщо(бпідґрунтя    ){
64              //Зачекайте на останню команду
65              очікуванн(рет, H ULL  );
66          }
67
68      }/ /виконати
```

У наведеному вище прикладі не виконується стандартне перенаправлення помилок (дескриптор файлу 2). Семантика цієї оболонки повинна полягати в тому, що всі прості команди надсилатимуть stderr туди ж. Приклад, наведений вище, можна змінити для підтримки переспрямування stderr.

## Вбудовані функції

Усі вбудовані функції, крім printenv, виконуються батьківським процесом. Причина цього полягає в тому, що ми хочемо, щоб setenv, cd і т. д. змінили батьківський стан. Якщо вони виконуються дитиною, зміни зникнуть, коли дитина вийде. Для цієї вбудованої функції викличте функцію всередині execute, а не розгалужуйте новий процес.

***Реалізація підстановкових знаків у Shell***

Жодна оболонка не обходиться без символів підстановки. Підстановки — це функція оболонки, яка дозволяє виконувати одну команду для кількох файлів, які відповідають символу підстановки.

Підстановочний знак описує імена файлів, які відповідають шаблону. Підстановка працює шляхом ітерації всіх файлів у поточному каталозі або каталозі, описаному в шаблоні, а потім як аргументи для команди ті імена файлів, які відповідають шаблону.

Загалом символ «*» відповідає 0 або більше символів будь-якого типу. Персонаж "?" відповідає одному символу будь-якого типу.

Щоб реалізувати підстановку, ви повинні спочатку перевести цей символ у регулярний вираз, який може оцінити бібліотека регулярних виразів.

Ми пропонуємо спочатку реалізувати простий випадок, коли ви розширюєте символи підстановки в поточному каталозі. У shell.y, де аргументи вставлені в таблицю, виконайте розширення.

shell.y:

**Перед:**

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

**Після:**

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

Далі вказана функція expandWildcardsIfNecessary(). У рядках з 4 по 7 буде вставлено аргумент, у якому аргумент arg не має «*» або «?» і негайно повертайся. Однак, якщо ці символи існують, він переведе підстановковий знак у регулярний вираз.

```
1  порожнеча expandWildcardsIfNecessary (char * arg)
2  {
3     //Returnifargne містить «*»або«?» якщо
4        (arghasnei». *ні « ?" ysestrchr ) {
5           Команда::_currentSimpleCommand > insertArgument (arg) ;
6           повернутися;
7     }
8
9     //1.Convertwildcardtoregularexpression //
10    Перетворити"*">".*"
11    //"?" >"."
12    //"." >"\." і інші, що вам потрібно
13    //Також додайте atthebeginningandtheendtomatch //
14    thebeginningandtheendofthe word
15    //Виділити достатньо місця для регулярного виразу
16    char * reg=(char*)malloc(2* strlen (arg) +10) ;
17    char * a= arg; //
18    char * p= reg; наприклад:
19    *p='^' ;p++;/ /matchbeinningoffline
20    поки (*a) {
21          якщо(*a=='*' ) {*p='.' ;p++;*p='*' ;p++;}
22          інакше (*a=='?' ) {*p='.' ;p++;}
23          інакше (*a=='.' ) {*p='\\' ;p++;*p='.' ;p++;}
24          інше {*p++=*a;p++;}
25          a++;
26    }
27    *p='$'; p++;*p=0; //matchendoflineandaddnullchar
28    //2.compileregularexpression.Seelab3src/regular.cc char
29          * expbuf =pegcomp (reg, наприклад, &e ) ;
30    якщо(expbuf ==NULL ) {
31          помилка("regcomp". ) ;
32          повернутися;
```

```
33      }
34      //3.Каталог списку та аргументи доданих записів //які
35      відповідають регулярному виразу
36      DIR * dir =opendir ("..");
37      якщо(dir ==NULL ){
38              помилка("відкрити каталог");
39              повернутися;
40      }
41      структурувати dirent * ent;
42      поки ((ent =readdir (dir))!= NULL){
43              //Перевірка збігів імен
44              якщо(regexec (ent >ім'я_файлу , &re, 0, 0, 0) == 0) {
45              //Додаток
46              Команда._currentSimpleCommand >
47              insertArgument (strdup (ent >ім'я_файлу ));
48              }
49      }
50      закрита (dir);
51  }
52
```

Основні переклади, які потрібно виконати з підстановки в регулярний вираз, наведено в наступній таблиці.

| *Символ підстановки* | *Регулярний вираз* |
|---|---|
| *"*"* | *«.*».* |
| *«?».* | *"."* |
| *"."* | *"\."* |
| Початок підстановки | *"^"* |
| Кінець підстановки | *"$"* |

У рядку 16 достатньо пам'яті виділено для регулярного виразу. Рядок 19. Вставте «^», щоб узгодити початок регулярного виразу з початком назви файлу, оскільки ми хочемо примусово збігати всю назву файлу. Рядки з 20 по 26 перетворюють символи підстановки в таблиці вище у відповідні еквіваленти регулярного виразу. Рядок 27 додає «$», що відповідає кінці регулярного виразу з кінцем імені файлу.

Рядки з 29 по 33 компілюють регулярний вираз у більш ефективне представлення, яке можна оцінити, і воно зберігає його в *xpbuf* . Рядок 41 відкриває поточний каталог і рядки з 42 по 48

перебирає всі імена файлів у поточному каталозі. Рядок 44 перевіряє, чи ім'я файлу відповідає регулярному виразу, і якщо воно відповідає дійсності, копія імені файлу буде додана до списку аргументів. Все це додасть до списку аргументів імена файлів, які відповідають регулярному виразу.

**Сортування записів довідника**

Оболонки, такі як bash, сортують записи, які відповідають символу підстановки. Наприклад, «echo *» відсортує всі записи в поточному каталозі. Щоб мати таку саму поведінку, вам доведеться змінити відповідність підстановки таким чином:

У рядку 5 створюється тимчасовий масив, який міститиме імена файлів, які відповідають символу підстановки. Початковий розмір масиву maxentries=20. Цикл while у рядку 7 повторює всі записи каталогу. Якщо вони збігаються, він вставить їх у тимчасовий масив. Рядки з 10 по 14 подвоїть розмір масиву, якщо кількість записів досягла максимальної межі. Рядок 20 відсортує записи за допомогою функції сортування на ваш вибір. Нарешті, рядки з 23 по 26 перебирають відсортовані записи в масиві та додають їх як аргумент у відсортованому порядку.

```
1
2   структурувати dirent  *  ent;
3   міжнар maxEntries  =2 0;
4   міжнар nЗаписи  =0 ;
5   char  * * array  =(c хар* *)м розподіли(maxEntries  zeof  (char* ));
6
7   поки  ((e  nt  =peaddir  (dir) )!=  ULL) {
8     //Перевірка збігів імен
9     якщо(regexec  (ent >d_ім'я  , buf  ) ) {
10          якщо(nЗаписи  maxEntries  ) {
11              maxEntries  *2
12              масив =realloc  rray  , maxEntries  *izeof  (char* ));
13              стверджувати(arra  !=NULL);
14          }
15          масив[nЗаписи  c_dup  (ent >d_ім'я  ) ;
16          nЗаписи + + ;
17    }
18  }
19  закрита  (dir) ;
20  sortArrayStrings  (array  , nЗаписи  ) ;//Використовуйте функцію сортування
21
22  //Додатки
23  для (int  я= 0  ; я  < n Записи  ;я + +){
24    Команда:_currentSimpleCommand  >
25    insertArgument  (array  [я] ));
26  }
27
28  безкоштовно(array  ) ;
```

### Підстановки та приховані файли

Ще одна особливість оболонок, таких як bash, полягає в тому, що символи підстановки за замовчуванням не відповідають прихованим файлам, які починаються з символу «.». У UNIX приховані файли починаються з “.” наприклад .login, .bashrc тощо.
Файли, які починаються з "." не повинно відповідати символу підстановки. Наприклад, "echo *" не відображатиме "." і «..».

Для цього оболонка додасть ім’я файлу, яке починається з «». тільки якщо підстановковий знак також має «.» на початку підстановки. Для цього оператор match if потрібно змінити таким чином:. Якщо ім’я файлу відповідає символу підстановки, то лише якщо ім’я файлу починається з '.' і символ підстановки починається з '.' потім додайте ім'я файлу як аргумент. Інакше, якщо ім’я файлу не починається з “.” потім додайте його до списку аргументів.

```
якщо (регулярний вираз (...) ) {
   якщо (ent>d_name[0] == '.') {
      if (arg[0] == '.')
         додати ім'я файлу до аргументів;
      }
    }
    інакше {
      додати ent>d_name до аргументів
    }
}
```



### Підкаталог підстановки

Підстановкові знаки також можуть відповідати каталогам у передані шляху:

Наприклад, «echo /p*/*a/b*/a*» відповідатиме не лише іменам файлів, а й підкаталогам у шляху.

Щоб узгодити підкаталоги, вам потрібно зіставити компонент за компонентом




Ви можете реалізувати стратегію підстановки таким чином:

Напишіть функцію expandWildcard(префікс, суфікс) де

> префікс Шлях, який уже розширено. Цьому не повинно бути символів підстановки. суфікс – частина шляху, що залишилася, яке не розширена. Він може мати символи підстановки.

Префікс буде вставлено як аргумент, коли суфікс порожній expandWildcard(prefix, suffix) спочатку викликається з порожнім префіксом і символом підстановки в суфіксі. expandWildcard буде викликатися рекурсивно для елементів, які збігаються в шляху.