Capítulo 5. Escrevendo seu próprio shell

Você realmente entende algo até programá-lo.
GRR

Introdução

O último capítulo abordou como usar um programa shell usando comandos UNIX. O shell é um programa que interage com o usuário através de um terminal ou recebe a entrada de um arquivo e executa uma sequência de comandos que são passados para o Sistema Operacional. Neste capítulo você aprenderá a escrever seu próprio programa shell.

Programas Shell

Um programa shell é um aplicativo que permite interagir com o computador. Em um shell o usuário pode executar programas e também redirecionar a entrada para vir de um arquivo e a saída para vir de um arquivo. Os shells também fornecem construções de programação como if, for, while, funções, variáveis etc. Além disso, os programas shell oferecem recursos como edição de linha, histórico, conclusão de arquivo, curingas, expansão de variáveis de ambiente e construções de programação. Aqui está uma lista dos programas shell mais populares no UNIX:

sh	Programa Shell. O programa shell original no UNIX.
csh	C Shell. Uma versão melhorada do sh.
tcsh	Uma versão do Csh que possui edição de linha.
ksh	Korn Shell. O pai de todos os shells avançados.
bash	O shell GNU. Pega o melhor de todos os programas shell. É atualmente o programa shell mais comum.

Além dos shells de linha de comando, também existem shells gráficos, como o Windows Desktop, MacOS Finder ou Linux Gnome e KDE que simplificam o uso de computadores para a maioria dos usuários. No entanto, esses shells gráficos não substituem os shells de linha de comando para usuários avançados que desejam executar sequências complexas de comandos repetidamente ou com parâmetros não disponíveis nos controles e diálogos gráficos amigáveis, mas limitados.

Partes de um programa Shell

A implementação do shell é dividida em três partes: o analisador, o executor e o shell.
Subsistemas.

O Parser é o

componente de software que lê a linha de comando, como “ls al”, e a coloca
em uma estrutura de dados chamada Tabela de Comandos que armazenará os comandos que serão
executado.

O executor

pegará a tabela de comandos gerada pelo analisador e para cada
SimpleCommand no array criará um novo processo. Também criará pipes se necessário
para comunicar a saída de um processo para a entrada do próximo. Além disso, ele irá
redirecionar a entrada padrão, a saída padrão e o erro padrão se houver algum redirecionamento.

A figura abaixo mostra uma linha de comando “A | B | C | D”. Se houver um redirecionamento como “< ***infile ”*** detectado pelo analisador, a entrada do primeiro SimpleCommand A é redirecionada de ***infile .*** Se houver um redirecionamento de saída como “> outfile ”, ele redireciona a saída do último SimpleCommand (D) para arquivar·

Se houver um redirecionamento para errfile como “>& errfile” o stderr de todos os SimpleCommand os processos serão redirecionados para errfile.

Subsistemas de Shell

Outros subsistemas que completam seu shell são:

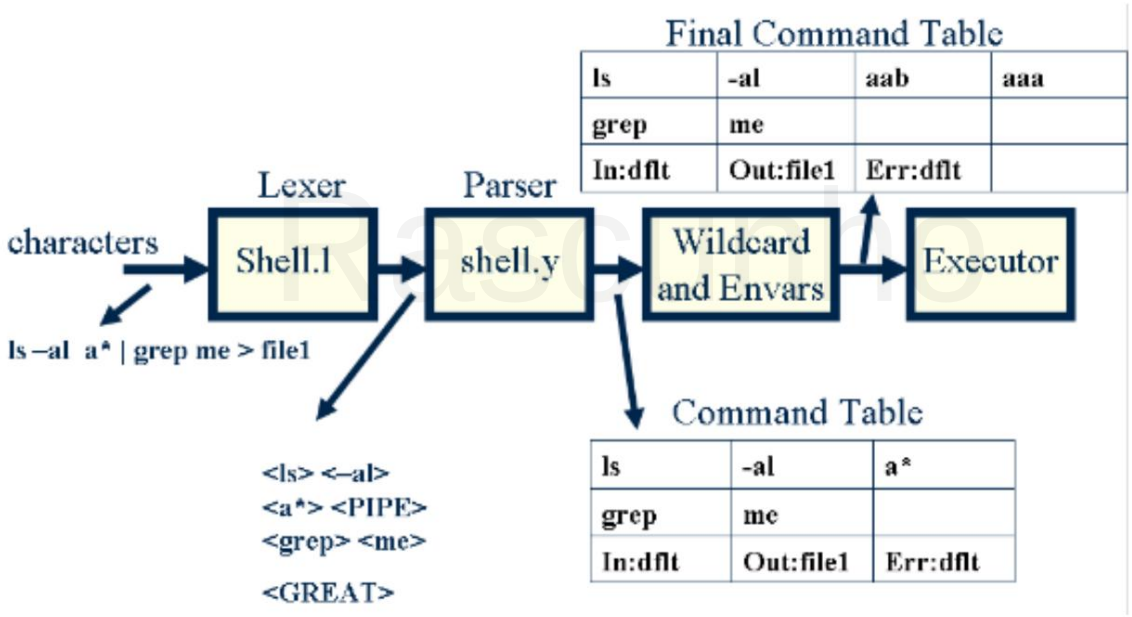
- Variáveis de ambiente: expressões do formulário ${VAR} são expandidas com o variável de ambiente correspondente. Além disso, o shell deve ser capaz de definir, expandir e imprimir variáveis de ambiente.
- Curingas: argumentos do formato a*a são expandidos para todos os arquivos que correspondem a eles em o diretório local e em vários diretórios.
- Subshells: Os argumentos entre `` (crases) são executados e a saída é enviada como entrada para o shell.

É altamente recomendável que você implemente seu próprio shell seguindo os passos deste capítulo.
Implementar seu próprio shell lhe dará uma ótima compreensão de como o shell
aplicações de interpretação e o sistema operacional interagem. Além disso, será um bom projeto para
mostre durante sua entrevista de emprego para futuros empregadores.

Usando Lex e Yacc para implementar o Parser Você
usará duas ferramentas UNIX para implementar seu parser: Lex e Yacc. Essas ferramentas são usadas para implementar compiladores, intérpretes e pré-processadores. Você não precisa saber compilador
teoria para usar essas ferramentas. Tudo o que você precisa saber sobre essas ferramentas será explicado em este capítulo.

Um analisador é dividido em duas partes: um analisador léxico ou lexer pega os caracteres de entrada e coloca os caracteres juntos em palavras chamadas tokens e um analisador que processa os
tokens de acordo com uma gramática e construir a tabela de comandos.

Aqui está um diagrama do Shell com o Lexer, o Parser e os outros componentes.



Os tokens são descritos em um arquivo shell.l usando expressões regulares. O arquivo shell.l é processado com um programa chamado lex que gera o analisador léxico.

As regras gramaticais usadas pelo analisador são descritas em um arquivo chamado shell.y usando sintaxe expressões que descrevemos abaixo. shell.y é processado com um programa chamado yacc que
gera um programa analisador. Tanto lex quanto yacc são comandos padrão no UNIX. Esses
comandos podem ser usados para implementar compiladores muito complexos. Para o shell, usaremos um subconjunto de Lex e Yacc para construir a tabela de comandos necessária ao shell.

Você precisa implementar a gramática abaixo em shell.l ***e shell.y*** para fazer nosso analisador interpretar as linhas de comando e fornecer ao nosso executor as informações corretas.

```
cmd [argumento]* [ | cmd [argumento]* ]*
[ [> nome do arquivo] [< nome do arquivo] [ >& nome do arquivo] [>> nome do arquivo] [>>& nome do arquivo] ]* [&]
```

Fig 4: Gramática Shell na forma BackusNaur

Esta gramática é escrita em um formato chamado "BackusNaur Form". Por exemplo cmd [arg] significa um comando, cmd , seguido por 0 ou mais argumentos, arg . A expressão [| cmd [arg]*]* representa os subcomandos de pipe opcionais onde pode haver 0 ou mais deles. A expressão [>filename] significa que pode haver 0 ou 1 redirecionamento >filename. O ***[&]*** no final significa que o caractere & é opcional.

Exemplos de comandos aceitos por esta gramática são:

ls-al

ls –al > fora

ls –al | classificar >& fora

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

A Mesa de Comando

A Command Table é uma matriz de ***estruturas SimpleCommand. Uma estrutura SimpleCommand*** contém membros para o comando e argumentos de uma única entrada no pipeline. O analisador também examinará a linha de comando e determinará se há alguma entrada ou saída redirecionamento baseado em símbolos presentes no comando (ou seja, < infile ou > outfile).

Aqui está um exemplo de um comando e da Tabela de Comandos que ele gera:

comando

ls al | grep me > arquivo1

Mesa de Comando

Matriz de comando simples:

0: eu		**tudo**	**NULO**
1: aderência		**meu**	**NULO**

Redirecionamento de E/S:

em: padrão	**fora: arquivo1**	**erro: padrão**

Para representar a tabela de comandos usaremos as seguintes classes: Command e **SimpleCommand.**

```
// Estrutura de dados do comando

// Descreve um comando simples e argumentos struct
SimpleCommand {
// Espaço disponível para argumentos atualmente pré-alocados int _numberOfAvailableArguments;


// Número de argumentos int _numberOfArguments;


// Matriz de argumentos char ** _arguments;


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


// Descreve um comando completo com vários pipes, se houver // e redirecionamento de entrada/
saída, se houver. struct Command { int

_numberOfAvailableSimpleCommands; int _numberOfSimpleCommands;
SimpleCommand ** _simpleCommands; char * _outFile;
char * _inputFile; char * _errFile; int _background;




prompt vazio(); imprimir vazio();
executar vazio(); limpar vazio();



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

Comando estático _currentCommand; Comando Simples
estático *_currentSimpleCommand; };
```

O construtor SimpleCommand::SimpleCommand constrói um comando simples vazio. O método SimpleCommand::insertArgument(char * argument) insere um novo argumento no SimpleCommand e amplia o array _arguments se necessário. Ele também garante que o último elemento seja NULL, pois isso é necessário para a chamada de sistema exec().

O construtor Command::Command() constrói um comando vazio que será **preenchido com o método Command::insertSimpleCommand(SimpleCommand *** simpleCommand). insertSimpleCommand também amplia o array _simpleCommands se necessário. As variáveis _outFile, _inputFile, _errFile serão NULL se nenhum redirecionamento foi feito, ou o nome do arquivo para o qual estão sendo redirecionadas.

As variáveis _currentCommand e _currentCommand são variáveis estáticas, ou seja, há apenas uma para toda a classe. Essas variáveis são usadas para construir o Command e o comando Simple durante a análise do comando.

As classes Command e SimpleCommand implementam a principal estrutura de dados que usaremos no shell.

Implementando o Analisador Lexical

O analisador léxico separa a entrada em tokens. Ele lerá os caracteres um por um da entrada padrão e formará um token que será passado para o analisador. O analisador léxico usa um arquivo shell.l que contém expressões regulares descrevendo cada um dos tokens. O analisador léxico lerá a entrada caractere por caractere e tentará corresponder a entrada com cada uma das expressões regulares. Quando uma string na entrada corresponder a uma das expressões regulares, ele executará o código {...} à direita da expressão regular. A seguir está uma versão simplificada de shell.l que seu shell usará:

```
/ * *
shell.l: analisador léxico simples para o shell. */


%{

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


%}

%%

\n { retornar
NOVA LINHA;
```

```
}

 [ \t] {
/* Descartar espaços e tabulações */ }


 ">" { retornar
ÓTIMO; }


 "<" { retornar
MENOS; }


 ">>" { retornar
ÓTIMOÓTIMO; }


 ">&" { return
GREATAMPERSAND; }


 “|” { retornar
PIPE; }


 “&” { retornar
E PERSPECTIVO; }



 [^ \t\n][^ \t\n]* { /* Suponha que os
nomes de arquivo tenham apenas caracteres alfabéticos */ yylval.string_val = strdup(yytext); return WORD; }



/* Adicione mais tokens aqui */

 . { /* Caractere
inválido na entrada */ return NOTOKEN; }



 %%
```

O arquivo shell.l para é passado através do gerado quando executando lex.yy.c. Este arquivo
lex implementa o scanner que o analisador usará para traduzir caracteres em tokens.

Aqui está o comando usado para executar o lex.

```
bash% lex shell.l
bater% ls
lex.aa.c
```

O arquivo lex.yy.cé um arquivo C que implementa o lexer para separar os tokens descritos em ***concha.l***

Há duas partes em shell.l. A parte superior se parece com isso:

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

Esta é uma parte que será inserida no topo do arquivo lex.yy.c diretamente sem modificação que inclui arquivos de cabeçalho e definições de variáveis que você usará no scanner. É aqui que você pode declarar variáveis que usará no seu analisador léxico.

A segunda parte delimitada por %% se parececom isto:

```
%%
\n {
retornar NOVA LINHA;
}
[ \t] {
/* Descartar espaços e tabulações */
}
">" {
retornar ÓTIMO;
}
[^ \t\n][^ \t\n]* {
/* Suponha que os nomes dos arquivos tenham apenas caracteres alfabéticos */
yylval.string_val = strdup(textoyy);
retornar PALAVRA;
}
%%
```

Esta parte contém as expressões regulares que definem os tokens formados pela tomada de caracteres da entrada padrão. Uma vez que um token é formado, ele será retornado ou, em alguns casos, descartado. Cada regra que define um token também tem duas partes:

expressão regular {
Ação
}

Por exemplo

\n {
retornar NOVA LINHA;
}

A primeira parte é uma expressão regular que descreve o token que esperamos corresponder. O ação é um pedaço de código C que o programador adiciona e que é executado assim que o token corresponde à expressão regular. No exemplo acima, quando o caractere de nova linha é encontrado, lex retornará a constante NEWLINE . Descreveremos mais tarde onde as constantesNEWLINE são definidos.

Aqui está um token mais complexo que descreve uma PALAVRA . ***Uma PALAVRA*** pode ser um argumento para uma comando ou o próprio comando.


```
[^ \t\n][^ \t\n]* {
/* Suponha que os nomes dos arquivos tenham apenas caracteres alfabéticos */
yylval.string_val = strdup(textoyy);
retornar PALAVRA;
}
```

A expressão em [...] corresponde a qualquer caractere que esteja dentro dos colchetes. A expressão [^...] corresponde a qualquer caractere que não esteja dentro dos colchetes. Portanto, [^ \t\n][^ \t\n]* descreve um token que começa com um caractere que não é um espaço, tabulação ou nova linha e é sucedido por zero ou mais caracteres que não sejam espaços, tabulações ou quebras de linha. O token correspondido está em uma variável chamado . Uma vez que uma palavra é correspondida, uma duplicata do token correspondido é atribuída a ***yylval.string_val*** yytext a seguinte declaração:

yylval.string_val = strdup(textoyy);

esta é a maneira como o valor do token é passado para o parser. Finalmente, a constante WORD é retornou ao analisador.

Adicionando novos tokens ao shell.l *O shell.l* descrito acima atualmente suporta um número reduzido de tokens. Como primeiro passo ao desenvolver seu shell, você precisará adicionar mais tokens à nova gramática que não são atualmente em shell.l . Veja a gramática na Figura 4 para ver quais tokens estão faltando e precisam ser ser adicionado ao shell.l Aqui estão alguns desses tokens:

```
">>" { retornar ÓTIMOÓTIMO; }
“|” { retornar PIPE;}
“&” {retornar E PERSPECTIVO}
Etc.
```

Adicionando os novos tokens ao shell.y Você adicionará os nomes dos tokens que criou na etapa anterior ao shell.y no %token seção:

```
%token NOTOKEN, ÓTIMO, NOVA LINHA, PALAVRA, ÓTIMOÓTIMO, PIPE,
E comercial etc.
```

Completando a gramática

Você precisa adicionar mais regras ao shell.y para completar a gramática do shell. O seguinte a figura separa a sintaxe do shell em diferentes partes que serão usadas para construir o gramática.

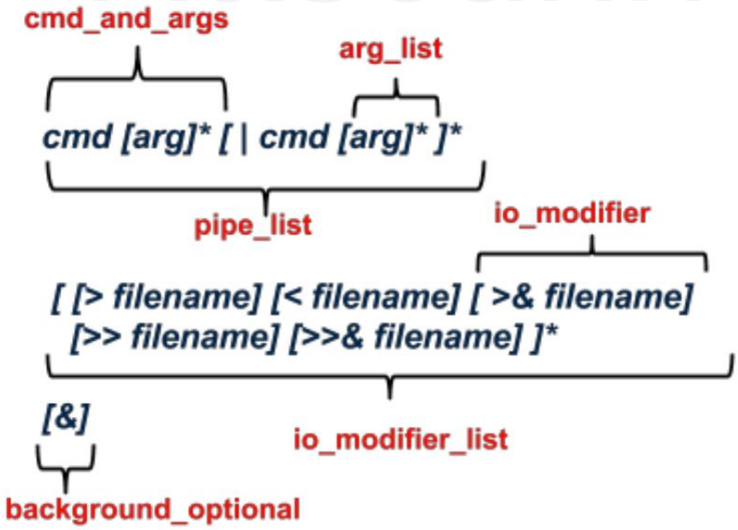


Figure 3. Shell Grammar labeld with the different parts.

Aqui está a gramática formada usando a rotulagem definida acima:

```
objetivo: lista_de_comandos;
```

```
arg_list:
        arg_list PALAVRA | /
          *vazio*/ ;

cmd_and_args:
        PALAVRA arg_list
                ;

lista_de_tubos: lista_de_tubos PIPE
        cmd_e_args | cmd_e_args
        ;

io_modificador:
        ÓTIMO ÓTIMO Palavra
        | GRANDE Palavra
        | GREATGREATMERSAND Palavra
        | GRANDE AMPERSAND Palavra
        | MENOS Palavra
        ;
io_modifier_list:
        lista_de_modificadores_io io_modifier
                | /*vazio*/
                ;
plano de fundo_opcional:
                E PERSPECTIVO
                | /*vazio*/
                ;
linha de comando:
lista_de_pipes lista_de_modificadores_io opção_de_fundo NOVA LINHA
        | NEWLINE /*aceitar linha de comando
        vazia*/ | erro
        NEWLINE{yyerrok;} /*recuperação de erro*/

lista_de_comandos:
                command_list command_line ;/
                * loop de comando*/
```


A gramática acima implementa o loop de comando na própria gramática.

O token de erro é um token especial usado para recuperação de erros. O erro analisará todos os tokens até que um token conhecido seja encontrado, como <NEWLINE>. O yerrok informa ao analisador que o erro foi recuperado.

O parser pega os tokens gerados pelo analisador léxico e verifica se eles seguem o . Ao de entrada descrita pelas regras gramaticais em verificar se a sintaxe da linha de comando shell.y segue a sintaxe, o parser executará ações ou pedaços de código C que você inserirá entre as regras gramaticais. Esses pedaços de código são chamados de ações e são delimitados por chaves { action; }.

Você precisa adicionar ações {…} na gramática para preencher a tabela de comandos.

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

;

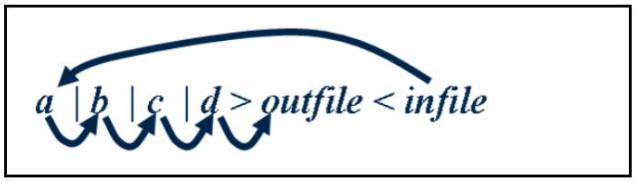
Criando Processos no Seu Shell

Comece criando um novo processo para cada comando no pipeline e fazendo o pai esperar pelo último comando. Isso permitirá executar comandos simples como “ls al”.


```
Comando::execute() { int ret;
 para
( int i = 0; i <
_numberOfSimpleCommands; i++ ) {
            direita = garfo(); if (direita
            == 0) {
//child
execvp(sCom[i]>_args[0], sCom[i]>_args); perror(“execvp”); _exit(1); } else if (ret < 0)
{ perror(“fork”); return; } // Shell pai
continue } // for if (!background)
            { //
espera pelo último processo waitpid(ret,
NULL); } }// executa
```

Redirecionamento de Pipe e Entrada/Saída em seu Shell

A estratégia para seu shell é fazer com que o processo pai faça todo o encanamento e redirecionamento antes de bifurcar os processos. Dessa forma, os filhos herdarão o redirecionamento. O pai precisa salvar entrada/saída e restaurá-la no final. Stderr é o mesmo para todos os processos



Nesta figura o processo a envia a saída para o pipe 1. Então b lê sua entrada do pipe 1 e envia sua saída para o pipe 2 e assim por diante. O último comando d lê sua entrada do pipe 3 e envie sua saída para outfile . A entrada de a vem de infile .

O código a seguir mostra como implementar esse redirecionamento. Alguma verificação de erro foi eliminado por simplicidade.


```
1 comando vazio::execute(){
2 // salvar entrada/saída
3 int tmpin=dup(0);
4 int tmpout=dup(1);
5
6 //defina a entrada inicial
7 você está redimido;
8 se (infile) {
9 fdin = abrir(infile,O_READ);
10 }
11 outro {
12 // Usar entrada padrão
13fdin=dup(tmpin);
14 }
15
16 int.
17 int fdout;
18 para (i = 0; i < num comandos simples; i++) {
19 // redirecionar entrada
20 dup2(fdin, 0);
21fechar(fdin);
22 //configurar saída
23 se (i == numsimplecommands1){
24 // Último comando simples
25 se(arquivo de saída){
```

```
26 fdout=open(outfile,â€¦â€¦);  27 } 28 else { 29 // Usar saída padrão 30
fdout=dup(tmpout);
31 } 32 } 33




34 else { 35 // Não último 36 //
comando simples 37 //criar pipe 38 int
fdpipe[2]; 39 pipe(fdpipe); 40 fdout=fdpipe[1]; 41
fdin=fdpipe[0]; 42 }// if/else 43




44 // Redirecionar saída 45 dup2 (fdout, 1); 46
close (fdout); 47


48 // Cria processo filho 49 ret = fork (); if (ret = 0) { 51
execvp (scmd [i]. args [0], scmd [i].
args); perror (â€œexecvpâ€);  53 _exit
(1); 54} 55} // para 56




57 //restaurar padrões de entrada/saída 58 dup2(tmpin,0);
59 dup2(tmpout,1); 60 close(tmpin);
61 close(tmpout); 62



63 if (!background) { 64 // Aguardar o último
comando 65 waitpid(ret, NULL); 66 } 67



68 } // executar
```


O método execute() é a espinha dorsal do shell. Ele executa os comandos simples em um processo separado para cada comando e realiza o redirecionamento.

As linhas 3 e 4 salvam o stdin e o stdout atuais em dois novos descritores de arquivo usando a função dup(). Isso permitirá que no final de execute() restaure o stdin e o stdout da maneira que estavam

no início de execute(). A razão para isso é que stdin e stdout (descritores de arquivo 0 e 1) será modificado no pai durante a execução dos comandos simples.

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

As linhas 6 a 14 verificam se há arquivo de redirecionamento de entrada na tabela de comando do formulário "command < infile". Se houver redirecionamento de entrada, ele abrirá o arquivo em infile e o salvará em ***fdin*** . Caso contrário, se não houver redirecionamento de entrada, ele criará um descritor de arquivo que se refere ao entrada padrão. No final deste bloco de instruções fdin será um descritor de arquivo que tem o entrada da linha de comando e que pode ser fechada sem afetar o programa shell pai.

```
6 //defina a entrada inicial
7 você está redimido;
8 se (infile) {
9 fdin = abrir(infile,O_READ);
10 }
11 outro {
12 // Usar entrada padrão
13fdin=dup(tmpin);
14 }
```


A linha 18 é o loop for que itera sobre todos os comandos simples na tabela de comandos. Isso ***for*** loop criará um processo para cada comando simples e executará o pipe conexões.

A linha 20 redireciona a entrada padrão para vir de fdin . Depois disso, qualquer leitura de stdin irá vêm do arquivo apontado por fdin . Na primeira iteração, a entrada do primeiro comando simples virá de fdin . ***fdin*** será reatribuído a um pipe de entrada mais tarde no loop. A linha 21 será fechada ***fdin,*** pois o descritor de arquivo não será mais necessário. Em geral, é uma boa prática fechar descritores de arquivo, desde que não sejam necessários, pois há apenas alguns disponíveis (normalmente 256 por padrão) para cada processo.

```
16 int.
17 int fdout;
18 para (i = 0; i < num comandos simples; i++) {
19 // redirecionar entrada
20 dup2(fdin, 0);
21fechar(fdin);
```

A linha 23 verifica se esta iteração corresponde ao último comando simples. Se for o caso, ele testará na linha 25 se há um redirecionamento de arquivo de saída do formato “command > outfile” e abrirá ***outfile*** e o atribuirá a fdout . Caso contrário, na linha 30 ele criará um novo descritor de arquivo que aponta para a entrada padrão. As linhas 23 a 32 garantirão que ***fdout*** seja um descritor de arquivo para a saída na última iteração.

```
23 //configurar saída 23 if (i ==
numsimplecommands 1){ // Último comando simples 25 if (outfile) { fdout = open
(outfile, â€¦â€¦);  27 } 28 else { // Usar saída padrão 30 fdout = dup
(tmpout); 31} 32 } 33




34 outro {...
```

As linhas 34 a 42 são executadas para comandos simples que não são o último. Para esses comandos simples, a saída será um pipe e não um arquivo. As linhas 38 e 39 criam um novo pipe. O novo pipe. Um pipe é um par de descritores de arquivo comunicados por um buffer. Qualquer coisa que seja escrita no descritor de arquivo fdpipe[1] pode ser lida de fdpipe[0]. Nas linhas 41 e 42, fdpipe[1] é atribuído a fdout e fdpipe[0] é atribuído a fdin.

A linha 41 fdin=fdpipe[0] pode ser o núcleo da implementação de pipes, pois faz com que a entrada fdin do próximo comando simples na próxima iteração venha de fdpipe[0] do comando simples atual.

```
34 else { 35 // Não último 36 //
comando simples 37 //criar pipe 38 int fdpipe[2]; 39
pipe(fdpipe); 40 fdout=fdpipe[1]; 41 fdin=fdpipe[0]; 42 }// if/
else 43
```

As linhas 45 redirecionam o stdout para ir para o objeto de arquivo apontado por fdout. Após essa linha, o stdin e o stdout foram redirecionados para um arquivo ou um pipe. A linha 46 fecha o fdout que não é mais necessário.

```
44 // Redirecionar saída 45 dup2 (fdout,
1); 46 close (fdout);
```

Quando o programa shell está na linha 48, os redirecionamentos de entrada e saída para o comando simples atual já estão definidos. A linha 49 bifurca um novo processo filho que herdará os descritores de arquivo 0,1 e 2 que correspondem a stdin, stdout e stderr, que são redirecionados para o terminal, um arquivo ou um pipe.

Se não houver erro na criação do processo, a linha 51 chama a chamada de sistema execvp() que carrega o executável para este comando simples. Se execvp for bem-sucedido, ele não retornará. Isso ocorre porque uma nova imagem executável foi carregada no processo atual e a memória foi sobrescrita, então não há nada para retornar.

```
48 // Cria processo filho 49 ret = fork (); if (ret =
0) { 51 execvp (scmd [i]. args [0],
scmd [i]. args); perror
("execvp"); 53 _exit (1); 54} 55} // para
```


A linha 55 é o fim do loop for que itera sobre todos os comandos simples.

Após a execução do loop for, todos os comandos simples estão sendo executados em seu próprio processo e estão se comunicando usando pipes. Como o stdin e o stdout do processo pai foram modificados durante o redirecionamento, as linhas 58 e 59 chamam dup2 para restaurar o stdin e o stdout para o mesmo objeto de arquivo que foi salvo em tmpin e tmpout. Caso contrário, o shell obterá a entrada do último arquivo para o qual a entrada foi redirecionada. Finalmente, as linhas 60 e 61 fecham os descritores de arquivo temporários que foram usados para salvar o stdin e o stdout do processo do shell pai.

```
57 //restaurar padrões de entrada/saída 58
dup2(tmpin,0); 59
dup2(tmpout,1); 60 close(tmpin);
61 close(tmpout);
```

Se o caractere de fundo "&" não foi definido na linha de comando, significa que o processo pai do shell deve esperar o último processo filho no comando terminar antes de imprimir o prompt do shell. Se o caractere de fundo "&" foi definido, significa que a linha de comando irá

execute assincronamente com o shell para que o processo do shell pai não espere o comando terminar e imprima o prompt imediatamente. Depois disso, a execução do comando é feita.

```
63 if (!background) { 64 // Aguardar o último
comando 65 waitpid(ret, NULL); 66 } 67



68 } // executar
```

O exemplo acima não faz redirecionamento de erro padrão (descritor de arquivo 2). A semântica deste shell deve ser que todos os comandos simples enviarão o stderr para o mesmo lugar. O exemplo dado acima pode ser modificado para suportar redirecionamento de stderr.

Funções Integradas

Todas as funções integradas, exceto printenv, são executadas pelo processo pai. O motivo para isso é que queremos que setenv, cd etc. modifiquem o estado do pai. Se forem executadas pelo filho, as alterações desaparecerão quando o filho sair. Para essas funções integradas, chame a função dentro de execute em vez de bifurcar um novo processo.

Implementando Wildcards no Shell Nenhum
shell está completo sem wildcards. Wildcards é um recurso do shell que permite que um único comando seja executado em vários arquivos que correspondem ao wildcard.

Um curinga descreve nomes de arquivo que correspondem ao curinga. Um curinga funciona iterando sobre todos os arquivos no diretório atual ou no diretório descrito no curinga e, em seguida, como argumentos para o comando, aqueles nomes de arquivo que correspondem ao curinga.

Em geral, o caractere “*” corresponde a 0 ou mais caracteres de qualquer tipo. O caractere ”?” corresponde a um caractere de qualquer tipo.

Para implementar um curinga, você deve primeiro traduzi-lo para uma expressão regular que uma biblioteca de expressões regulares possa avaliar.

Sugerimos implementar primeiro o caso simples em que você expande curingas no diretório atual. Em shell.y, onde argumentos são inseridos na tabela, faça a expansão.

concha.y:

Antes:

argumento: PALAVRA {

Comando::_currentSimpleCommand>insertArgument($1); } ;

Depois:

argumento: WORD

{ expandWildcardsIfNecessary($1); } ;

A função expandWildcardsIfNecessary() é dada a seguir. As linhas 4 a 7 inserirão o argumento o argumento arg não tem “*” ou “?” e retornarão imediatamente. No entanto, se esses caracteres existirem, então ele traduzirá o curinga para uma expressão regular.

```
1 void expandWildcardsIfNecessary(char * arg) 2 { 3 // Retorna se arg não contiver “*” ou “?”
4 if (arg não
tem “*” nem “?” (use strchr) ) { 5 Command::_currentSimpleCommand>insertArgument(arg); 6
return; 7 } 8 9 // 1. Converta curinga em expressão regular 10 // Converta “*” > “.*” 11 // “?” > “.” 12 // “.” >
“\.” e outros que você precisar 13 // Adicione também ^ no início e $ no final para corresponder 14 // ao início e ao fim da
palavra. 15 // Aloque espaço suficiente
para a expressão
regular
16 char * reg = (char*)malloc(2*strlen(arg)+10); 17 char * a = arg; 18 char * r = reg; 19 *r = '^'; r++; //
corresponda ao início da linha 20 while (*a) { 21 22 23 24
25 26 } 27 *r='$'; r++; *r=0;// corresponda
ao final da linha e adicione nulo char 28 // 2. compile a expressão regular. Veja
lab3src/regular.cc 29 char * expbuf = regcomp( reg, â€¦ );  30 if (expbuf==NULL) { 31 perror(“regcomp”); 32 return;

se (*a == '*') { *r='.'; r++; *r='*'; r++; } senão se (*a == '?') { *r='.' r++;} senão se (*a == '.')
{ *r='\\'; r++; *r='.'; r++;} senão { *r=*a; r++;} a++;
```

```
33 }
34 // 3. Liste o diretório e adicione como argumentos as entradas
35 // que correspondem à expressão regular
36 DIR * dir = opendir(".");
37 se (dir == NULL) {
38 perror(“oopendir”);
39 retorno;
40 }
41 struct diga* ent;
42 enquanto ( (ent = readdir(dir))!= NULL) {
43                 // Verifique se o nome corresponde
44                 se (regexec(ent>d_name, re ) ==0 ) {
45                 // Adicionar argumento
46                 Comando::_currentSimpleCommand>
47                 inserirArgumento(strdup(ent>d_name));
48                 }
49 }
50 fechadoir(dir);
51 }
52
```

As traduções básicas a serem feitas de um curinga para uma expressão regular estão no seguinte mesa.

Caractere curinga	***Expressão regular***
“*”	**“.*”**
“?”	**“.”**
“.”	**“\.”**
Início do curinga	**“^”**
Fim do Wildcard	**“$”**

Na linha 16, há memória suficiente alocada para a expressão regular. Linha 19. Insira o “^” para combine o início da expressão regular com o início do nome do arquivo, pois
quer forçar uma correspondência de todo o nome do arquivo. As linhas 20 a 26 convertem os caracteres curinga em a tabela acima para os equivalentes correspondentes da expressão regular. A linha 27 adiciona o “$” que corresponde ao final da expressão regular com o final do nome do arquivo.

As linhas 29 a 33 compilam a expressão regular em uma representação mais eficiente que pode ser avaliado e o armazena em expbuf. A linha 41 abre o diretório atual e as linhas 42 a 48

itera sobre todos os nomes de arquivo no diretório atual. A linha 44 verifica se o nome do arquivo corresponde à expressão regular e, se for verdadeiro, uma cópia do nome do arquivo será adicionada à lista de argumentos. Tudo isso adicionará os nomes de arquivo que correspondem à expressão regular à lista de argumentos.

Classificando entradas de

diretório Shells como bash classificam as entradas correspondidas por um curinga. Por exemplo, “echo *” listará todas as entradas no diretório atual classificadas. Para ter o mesmo comportamento, você terá que modificar a correspondência de curinga da seguinte forma:

A linha 5 cria um array temporal que manterá os nomes de arquivo correspondidos pelo curinga. O tamanho inicial do array é maxentries=20. O loop while na linha 7 itera sobre todas as entradas do diretório. Se elas corresponderem, ele as inserirá no array temporal. As linhas 10 a 14 dobrarão o tamanho do array se o número de entradas tiver atingido o limite máximo. A linha 20 classificará as entradas usando a função de classificação de sua escolha. Finalmente, as linhas 23 a 26 iteram sobre as entradas classificadas no array e as adicionam como argumento em ordem classificada.

```
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 // Verifique se o nome corresponde 9 if
(regexec(ent>d_name, expbuf) ) { 10 if (nEntries ==
maxEntries) { maxEntries *=2; array = realloc(array, maxEntries*sizeof(char*));
assert(array!=NULL); } array[nEntries]= strdup(ent>d_name); nEntries++;
11
12
13
14
15
16
17 } 18 } 19
closedir(dir);
20 sortArrayStrings(array,
nEntries); // Use qualquer função de classificação 21

22 // Adicione argumentos 23
para (int i = 0; i < nEntradas; i++) {
24 Comando::_currentSimpleCommand> 25 insertArgument(array[i]));
26 } 27


28 livre(matriz);
```

Curingas e Arquivos Ocultos Outro

recurso de shells como o bash é que curingas por padrão não corresponderão a arquivos ocultos que começam com o caractere “.”. No UNIX, arquivos ocultos começam com “.” como .login, .bashrc etc. Arquivos que começam com “.” não devem ser correspondidos com um curinga. Por exemplo, “echo *” não exibirá “.” e “..”.

Para fazer isso, o shell adicionará um nome de arquivo que começa com “.” somente se o curinga também tiver um “.” no começo do curinga. Para fazer isso, a instrução match if precisa ser modificada da seguinte maneira:. Se o nome do arquivo corresponder ao curinga, então somente se o nome do arquivo começar com '.' e o curinga começar com '.' então adicione o nome do arquivo como argumento. Caso contrário, se o nome do arquivo não começar com “.” então adicione-o à lista de argumentos.

```
se (regexec (…) ) { se
(ent>d_name[0] == '.') { se (arg[0] == '.')
adicionar nome do arquivo
aos argumentos; } } senão { adicionar ent>d_name
aos

argumentos } }
```


Curingas de subdiretório

Os curingas também podem corresponder a diretórios dentro de um caminho:

Por exemplo, “echo /p*/*a/b*/aa*” corresponderá não apenas aos nomes dos arquivos, mas também aos subdiretórios no caminho.

Para corresponder aos subdiretórios, você precisa corresponder componente por componente

Você pode implementar a estratégia curinga da seguinte maneira.

Escreva uma função expandWildcard(prefix, suffix) onde prefix O
caminho que já foi expandido. Não deve ter curingas. suffix – A parte restante do caminho que ainda não foi expandida. Pode ter curingas.

O prefixo será inserido como um argumento quando o sufixo estiver vazio.
expandWildcard(prefix, suffix) é inicialmente invocado com um prefixo vazio e o curinga em suffix. expandWildcard será chamado recursivamente para os elementos que correspondem no caminho.