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. A casca 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 ao sistema operacional. Neste capítulo você aprenderá como 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 um arquivo e a saída para um arquivo. Shells também fornecem construções de programação como if, for, while, funções, variáveis Além disso, os programas shell oferecem recursos como edição de linha, histórico, preenchimento de arquivo, curingas, expansão de variáveis de ambiente e construções de programação. Aqui está uma lista dos programas shell mais populares no UNIX:

eh	Programa Shell. O programa shell original no UNIX.
csh	Casca C. Uma versão melhorada do sh.
tcsh	Uma versão do Csh que possui edição de linha.
ksh	Concha de Korn. O pai de todos os shells avançados.
bash	O shell GNU. Leva o melhor de todos os programas shell. Isso 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 nas caixas de diálogo e controles gráficos amigáveis, mas limitados.

Partes de um programa Shell

A implementação do shell é dividida em três partes: O Analisador, O Executor e Shell **Subsistemas.**

O Analisador

O Analisador é o componente de software que lê a linha de comando como “ls al” e a coloca em uma estrutura de dados chamada Command Table que irá armazenar os comandos que serão executado.

O Executor O

executor pegará a tabela de comandos gerada pelo analisador e para cada SimpleCommand no array criará um novo processo. Também criará, se necessário, tubos comunicar a saída de um processo à entrada do próximo. Além disso, será redirecione 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 ***no arquivo .*** Se houver um redirecionamento de saída como “>outfile ”, ele redireciona a saída do último SimpleCommand (D) para arquivo de saída·

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

Subsistemas Shell

Outros subsistemas que completam seu shell são:

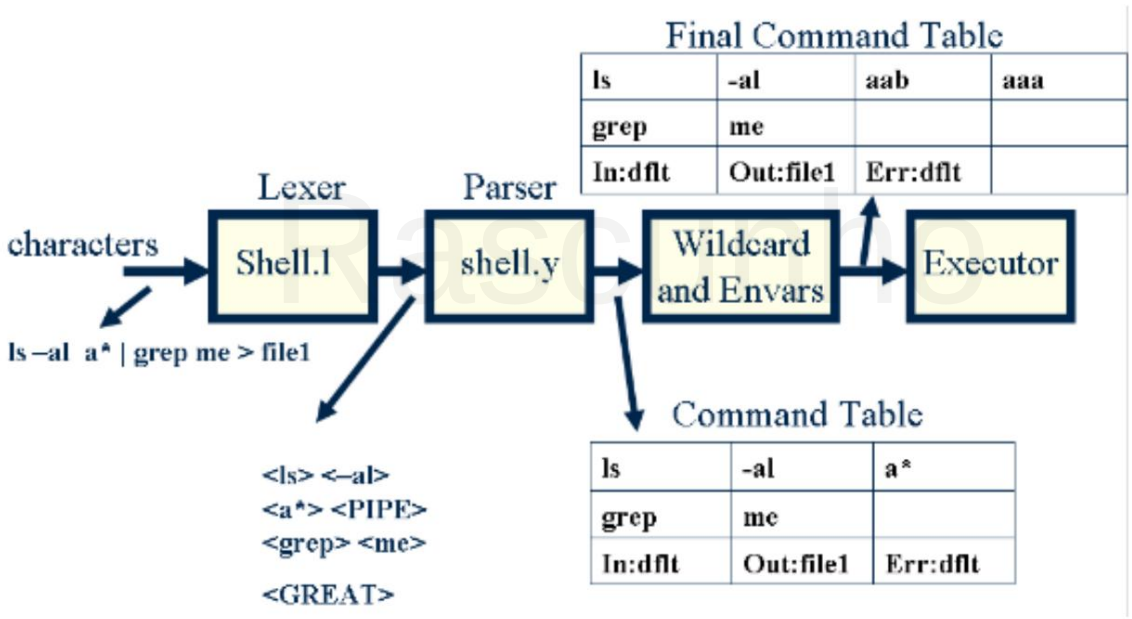
- Variáveis de ambiente: expressões no formato ${VAR} são expandidas com o variável de ambiente correspondente. Além disso, o shell deve ser capaz de definir, expandir e ambiente de impressão vars.
- Curingas: argumentos no formato a*a são expandidos para todos os arquivos que correspondem a eles o diretório local e em vários diretórios.
- Subshells: 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 as etapas deste capítulo. Implementar seu próprio shell lhe dará uma compreensão muito boa de como o shell aplicativos interpretadores 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 analisador Você
usará duas ferramentas UNIX para implementar seu analisador: Lex e Yacc. Essas ferramentas são usadas para implementar compiladores, interpretadores e pré-processadores. Você não precisa conhecer 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 Lexical ou Lexer pega os caracteres de entrada e reúne os caracteres 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 demais componentes.



Os tokens sãodescritos 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 [arg]* [| cmd [argumento]*]*

[[> nome do arquivo] [< nome do arquivo] [>& nome do arquivo] [>> nome do arquivo] [>>& nome do arquivo]]* [&]

Fig 4: Gramática Shell no formulário BackusNaur

Esta gramática é escrita em um formato denominado "Formulário BackusNaur". 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 [>nome do arquivo] significa que pode haver 0 ou 1 redirecionamentos >nome do arquivo. 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 >&

awk –f x.awk | classificar –u < arquivo de entrada > arquivo de saída &

A Tabela de Comando

A Tabela de Comandos é uma matriz de ***estruturas SimpleCommand. Uma estrutura SimpleCommand*** contém membros para o comando e argumentos de uma única entrada no pipeline. O 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 a tabela de comandos que ele gera:

comando

é tudo | grep-me > arquivo1

Tabela de Comandos

Matriz SimpleCommmand:

0: ls		**tudo**	**NULO**
1: grep		**meu**	**NULO**

Redirecionamento de IO:

em: padrão	**saída: arquivo1**	**errar: padrão**

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

```
//Estrutura de dados de 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 Comando { 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 );

Comando estático _currentCommand; static
SimpleCommand *_currentSimpleCommand; };
```

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

O construtor Command::Command() constrói um comando vazio que será **preenchido com o método Command::insertSimpleCommand(SimpleCommand *** simpleCommand). insertSimpleCommand também amplia a matriz _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, existe apenas uma para toda a classe. Essas variáveis são usadas para construir o comando Command e Simple durante a análise do comando.

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

Implementando o Analisador Lexical

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

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


%{

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

%}

%%

 \n {retorna
NOVA LINHA;
```

```
}

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


 ">" { retornar
ÓTIMO; }


 "<" { retornar
MENOS; }


 ">>" { return
ÓTIMO; }


 ">&" { return
GREATAMPERSAND; }


 “|” { retornar
PIPE; }


 “&” { return
E comercial; }


 [^ \t\n][^ \t\n]* { / * Suponha
que os nomes dos arquivos 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 é passado para o Lex. Lex gera um arquivo chamado lex.yy.c. Este ficheiro 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
bash% ls
lex.yy.c
```

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

Existem duas partes no shell.l. A parte superior fica assim:

```
%{
#include <string.h>
#include "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 aí que você pode declarar variáveis que usará em seu lexer.

A segunda parte delimitada por %% tem esta aparência:

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

Esta parte contém as expressões regulares que definem os tokens formados tomando o caracteres da entrada padrão. Assim que um token for formado, ele será devolvido ou, em alguns casos, descartado. Cada regra que define um token também possui 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 é executado assim que o token corresponde à expressão regular. No exemplo acima, quando o caractere 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 um WORD . ***UmaPALAVRA*** pode ser um argumento para uma comando ou o próprio comando.


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

A expressão em [...] corresponde a qualquer caractere que esteja entre colchetes. A expressão [^...] corresponde a qualquer caractere que não esteja entre colchetes. Portanto, [^ \t\n][^ \t\n]* descreve um token que começa com um caractere que não é espaço, tabulação ou nova linha e é sucedido por zero ou mais caracteres que não sejam espaços, tabulações ou novas linhas. O token correspondido está em uma variável chamado . Depois que uma palavra é correspondida, uma duplicata do token correspondente é atribuída a ***yylval.string_val*** yytexta seguinte instrução:

yylval.string_val = strdup(yytext);

é assim que o valor do token é passado para o analisador. Finalmente, a constante WORDé retornado ao analisador.

Adicionando novos tokens ao shell.l

O *shell.l* descrito acima atualmente suporta um número reduzido de tokens. Como o 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:

```
">>" { return ÓTIMO; }
“|” { retornar PIPE;}
“&” {retorna E comercial}
Etc.
```

Adicionando os novos tokens ao shell.y

Você adicionará os nomes dos tokens criados na etapa anterior ao shell.y no %token seção:

```
%token NOTOKEN, GREAT, NEWLINE, WORD, GREATGREAT, PIPE,
E comercial etc.
```

Completando a gramática

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

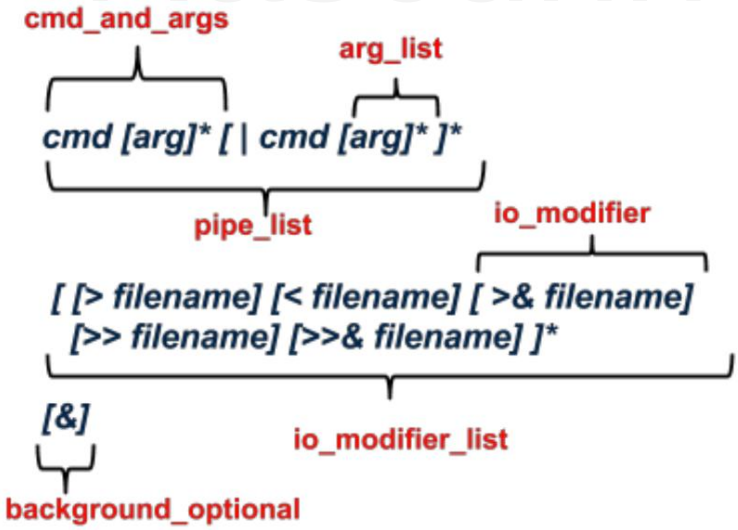


Figure 3. Shell Grammar labeld with the different parts.

Aqui está a gramática formada usando a rotulagem definida acima:

```
objetivo: lista_de_comandos;
```

```
lista_arg:
        lista_arg
        PALAVRA

| /*vazio*/ ; cmd_and_args:
        WORD lista_arg
                ;
pipe_list:
        pipe_list PIPE cmd_and_args
        | cmd_and_args
        ;

io_modificador:
        GRANDE GRANDE Palavra
        | GRANDE Palavra
        | GREATGREATAMPERSAND Palavra
        | GRANDE AMPERSAND Palavra
        | MENOS Palavra
        ;
io_modifier_list:
        io_modifier_list io_modifier | /
                *vazio*/
                ;
plano de fundo_opcional:
                EAMPERSAND
                | /*vazio*/
                ;
linha_de_comando:
pipe_list io_modifier_list background_opt NEWLINE
        | NEWLINE /*aceita linha cmd vazia*/ | erro
        NEWLINE{yyerrok;} /
        *recuperação de erros*/

lista_de_comandos:
                lista_de_comandos linha_de_comando ;/
                * 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>. yerrok informa ao analisador que o erro foi recuperado.

O analisador 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 analisador executará ações ou pedaços de código C que você inserirá entre as regras gramaticais. Esses trechos 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:
lista_arg:
lista_arg WORD { currSimpleCmd>insertArg($2); } | /
vazio/
;

Criando processos em seu shell

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

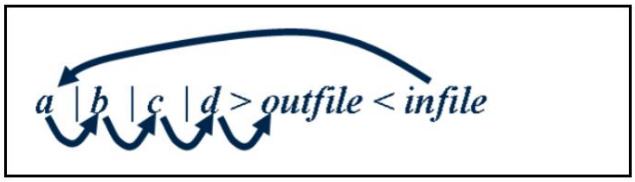

```
Command::execute() { int
 ret;
for ( int i = 0; i <
_numberOfSimpleCommands; i++ ) {
        ret = fork(); if (ret ==
        0) {
//child
execvp(sCom[i]>_args[0], sCom[i]>_args); perror(“execvp”); _exit(1); }
else if (ret < 0) { perror(“fork”) ;
return; } // Shell pai

continua } // for if (!background) { //
espera pelo último processo
waitpid(ret, NULL); } }//

        executa
```

Redirecionamento de canal e entrada/saída em seu shell

A estratégia para o seu shell é fazer com que o processo pai faça toda a tubulação e redirecionamento antes de bifurcar os processos. Desta forma, os filhos herdarão o redirecionamento. O pai precisa salvar a 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 tubo 1 . Entãob lê sua entrada do tubo 1 e envia sua saída para o tubo 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 do infile .

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


```
1 void Command::executar(){
2 //salvar entrada/saída
3 int tmpin=dup(0);
4 int tmpout=dup(1);
5
6 //defina a entrada inicial
7 intfdin;
8 se (arquivo) {
9 fdin = abrir(arquivo,O_READ);
10}
11 mais {
12 // Usar entrada padrão
13fdin=dup(tmpin);
14}
15
16intret;
17 intfdout;
18 for(i=0;i<numsimplecommands; i++) {
19 //redirecionar entrada
20 dup2(fdin, 0);
21fechar(fdin);
22 //configuração da saída
23 se (i == numsimplecommands1){
24 // Último comando simples
25 se(arquivo de saída){
```

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




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




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

48 // Cria processo filho 49 ret=fork(); 50
if(ret==0) { 51
execvp(scmd[i].args [0],
scmd[i].args); 52 perror(â€œexecvpâ€);  53 _exit(1); 54 } 55 } //para
56




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

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


68} //executa
```


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 stdout atuais em dois novos descritores de arquivo usando a função dup(). Isso permitirá que no final do execute() restaure o stdin e o stdout como estavam

no início de execute(). A razão para isto 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 existe arquivo de redirecionamento de entrada na tabela de comandos do formulário "comando <arquivo". Se houver redirecionamento de entrada, ele abrirá o arquivo no infile e o salvará em ***encontrar .*** Caso contrário, se não houver redirecionamento de entrada, será criado um descritor de arquivo que se refere ao entrada padrão. No final deste bloco de instruções, fdin estará um descritor de arquivo que possui o entrada da linha de comando e que pode ser fechada sem afetar o programa shell pai.

```
6 //defina a entrada inicial
7 intfdin;
8 se (arquivo) {
9 fdin = abrir(arquivo,O_READ);
10}
11 mais {
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. Esse ***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 do stdin será vem do arquivo apontado por fdin . Na primeira iteração, a entrada do primeiro comando simples virá de fdin . ***fdin*** será reatribuído a um canal de entrada posteriormente 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 arquivos, desde que não sejam necessários, pois há apenas alguns disponíveis (normalmente 256 por padrão) para cada processo.

```
16intret;
17 intfdout;
18 for(i=0;i<numsimplecommands; i++) {
19 //redirecionar entrada
20 dup2(fdin, 0);
21fechar(fdin);
```

A linha 23 verifica se esta iteração corresponde ao último comando simples. Se for esse o caso, ele irá testar na Linha 25 se há um redirecionamento do arquivo de saída no formato “command > outfile” e abrir ***outfile*** e atribuí-lo a fdout . Caso contrário, na linha 30 será criado 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 //saída de configuração 23 if (i
== numsimplecommands1){ 24 // Último comando simples
25 if(outfile){ 26 fdout=open( outfile,â€¦€¦); 27 } 28
else { 29 // Use a saída padrão 30
fdout=dup(tmpout); 31 } 32 } 33




34 mais{...
```

As linhas 34 a 42 são executadas para comandos simples que não são os últimos. Para esses comandos simples, a saída será um canal e não um arquivo. As linhas 38 e 39 criam um novo tubo. O novo tubo. Um pipe é um par de descritores de arquivo comunicados por meio de um buffer. Qualquer coisa escrita no descritor de arquivo fdpipe[1] pode ser lida em 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 é o
último 36 //comando simples 37 //cria
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 esta linha, o stdin e o stdout foram redirecionados para um arquivo ou canal. A linha 46 fecha o fdout que não é mais necessário.

```
44 // Redireciona a 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 canal.

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 o execvp for bem-sucedido, ele não retornará. Isso ocorre porque uma nova imagem executável foi carregada no processo atual e a memória foi sobrescrita, portanto não há nada para onde retornar.

```
48 // Cria processo filho 49 ret=fork(); 50
if(ret==0) { 51
execvp(scmd[i].args [0],
scmd[i].args); 52 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 são executados em seu próprio processo e se comunicam por meio de pipes. Como o stdin e o stdout do processo pai foram modificados durante o redirecionamento, as linhas 58 e 59 chamam dup2 para restaurar stdin e 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 arquivos temporários que foram usados para salvar o stdin e o stdout do processo shell pai.

```
57 // restaura os 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 aguardar a conclusão do último processo filho no comando antes de imprimir o prompt do shell. Se o caractere de fundo "&" foi definido, significa que a linha de comando será

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

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


68} //executa
```

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

Funções integradas

Todas as funções integradas, exceto printenv, são executadas pelo processo pai. A razão 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 esta função construída, chame a função dentro de execute em vez de bifurcar um novo processo.

Implementando curingas no Shell

Nenhum shell está completo sem curingas. Curingas é um recurso de shell que permite que um único comando seja executado em vários arquivos que correspondam ao curinga.

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

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

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

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

concha.y:

Antes:

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

Depois: ***argumento:***
WORD { expandWildcardsIfNecessary($1); } ;

A função expandWildcardsIfNecessary() é fornecida a seguir. As linhas 4 a 7 irão inserir o argumento o argumento arg não possui “*” ou “?” e retorne imediatamente. No entanto, se esses caracteres existirem, o curinga será traduzido em uma expressão regular.

```
1 void expandWildcardsIfNecessary(char * arg) 2 { 3 // Retorna se
arg não
contém “*” ou “?” 4 if (arg não tem “*” nem “?” (use strchr) ) { 5
Command::_currentSimpleCommand>insertArgument(arg); 6 return ; 7 } 8
9 // 1. Converte curinga em expressão regular 10 // Converte “*” > “.*” 11 // “?” > “.” 12 //
“.” > “\. ” e outros você
precisa de
13 //
Adicione também ^ no início e $ no final para corresponder a 14 // o
início e o 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++; // corresponde ao início da linha
20 while ( *a) { 21 22 23 24 25 26 } 27 *r='$'; r++; *r=0;// corresponde
ao final da linha e adiciona nulo char 28 // 2. compilar expressão regular.
Consulte lab3src/regular.cc 29 char * expbuf = regcomp( reg, €¦ );  30 if
(expbuf==NULL) { 31
perror(“regcomp ”); 32 retorno;


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

```
33 }
34 // 3. Lista o diretório e adiciona como argumentos as entradas
35 // que correspondem à expressão regular
36 DIR * dir = opendir(“.”);
37 se (dir == NULO) {
38 perror(“oopendir”);
39 retorno;
40 }
41 struct dirent* ent;
42 while ( (ent = readdir(dir))!= NULL) {
43                 //Verifica se o nome corresponde
44                 if (regexec(ent>d_name, re ) ==0 ) {
45                 //Adiciona argumento
46                 Comando::_currentSimpleCommand>
47                 insertArgument(strdup(ent>d_name));
48                 }
49 }
50 fechado(dir);
51}
52
```

As traduções básicas a serem feitas de um curinga para uma expressão regular estão a seguir mesa.

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

Na linha 16 é alocada memória suficiente para a expressão regular. Linha 19. Insira o “^” para combine o início da expressão regular com o início do nome do arquivo, pois deseja forçar uma correspondência de todo o nome do arquivo. As linhas 20 a 26 convertem os caracteres curinga em tabela acima aos 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 armazena-o em expbuf. A linha 41 abre o diretório atual e as linhas 42 a 48

itera sobre todos os nomes de arquivos 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 dos arquivos que correspondem à expressão regular à lista de argumentos.

Classificando entradas de

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

A linha 5 cria uma matriz temporal que conterá os nomes dos arquivos correspondentes ao curinga. O tamanho inicial da matriz é maxentries=20. O loop while na linha 7 itera sobre todas as entradas do diretório. Se eles corresponderem, ele os inserirá no array temporal. As linhas 10 a 14 dobrarão o tamanho do array se o número de entradas atingir 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 na matriz e as adicionam como argumento na ordem classificada.

```
1
2 structdirent*ent; 3int maxEntries
= 20; 4 int nEntries = 0; 5 char **
array = (char**)
malloc(maxEntries*sizeof(char*)); 6

7 while ( (ent = readdir(dir))!= NULL) { 8 // Verifica se o
nome corresponde a 9 if
(regexec(ent>d_name, expbuf) ) { 10 if (nEntries ==
maxEntries) { maxEntries *=2; array = realloc(array,
11                  maxEntries*sizeof(char* ));
12                  assert(array!=NULL); } array[nEntries]= strdup(ent>d_name);
13                  nEntries++;
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 < nEntries; i++) {
24 Comando::_currentSimpleCommand> 25
insertArgument(array[i])); 26 } 27


28 grátis(matriz);
```

Curingas e arquivos ocultos Outro

recurso de shells como o bash é que os curingas, por padrão, não corresponderão a arquivos ocultos que começam com o caractere “.”. No UNIX, os arquivos ocultos começam com “.” como .login, .bashrc etc. Arquivos que começam com “.” não deve ser combinado 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 início do curinga. Para fazer isso, a instrução match if deve ser modificada da seguinte maneira: Se o nome do arquivo corresponder ao curinga, somente se o nome do arquivo começar com '.' e o curinga começa com '.' em seguida, adicione o nome do arquivo como argumento. Caso contrário, se o nome do arquivo não começar com “.” em seguida, adicione-o à lista de argumentos.

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


{ 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 combinar subdiretórios você precisa combinar 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 deveria ter curingas. sufixo – A parte restante do caminho que ainda não foi expandida. Pode ter curingas.

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