1　目的

本レポートの目的は、シェルの内部の動作原理を理解することである。シェルとは、ユーザが入力したコマンドを解釈し、コマンドに応じて必要な処理を行うプログラムである。このシェルの機能を理解するために、代表的なシェルの機能であるリダイレクトやパイプを実装することで、シェルの動作原理に関して理解を深めることを目的とする。また、他人が作成した洗練されたコードを読むことで、プログラムの作成方法やコードの見やすさについて学ぶことも目的とする。

2　設計

今回のシェルはパイプ、リダイレクトの実装を主としている。そのため、プログラムの流れは以下のように設計した。

(1) ユーザが入力したコマンドを分割する
(2) 入力されたコマンドをすべて見る
(3) コマンドのみの時は、通常のコマンドを実行する
(4) リダイレクトがあるときは、リダイレクトを実行する
(5) パイプとリダイレクト両方があるときは、両方に対応するように処理をする。

3　実装

実装したプログラムは長いためすべては付録に記載する。ここでは、cd コマンドの処理とパイプとリダイレクトの処理を行っている関数について説明する。

3.1　cd コマンドの処理

次に示すコードが cd コマンドの処理を行っている関数である。

ソースコード 1　cd コマンドの処理

```
1  void cd(char **argv){
2    char path[BUFSIZE];
3    getcwd(path,BUFSIZE);
4    if(strstr(argv[1],"/")==NULL){
5      strcat(path,"/");
6      strcat(path,argv[1]);
7    }
8    if(chdir(path)==-1){
9      printf("no file\n");
10     exit(1);
11   }
12 }
```

引数として、標準入力で受け取った文字列を渡す。cd の後ろは相対パスが入力されると考え絶対パス表示にするため,getcwd を利用している。ユーザの入力の先頭に/がないと絶対パスの指定がうまくいかないため、/がない場合は/を追加して絶対パスを指定できるようにしている。

3.2　パイプリダイレクト処理

パイプリダイレクト処理は関数が長いため、付録で記載する。

(1) 入力された文字列をすべて確認し、リダイレクトパイプの場所を記録する
(2) 通常のコマンドの場合はそのままコマンドを実行する
(3) 逆のリダイレクトがあるときは、逆のリダイレクトを実行する
(4) リダイレクトがあるときは、リダイレクトを実行する
(5) パイプとリダイレクト両方があるときは、両方に対応するように処理をする。そうでない場合はパイプのみを実行する。

4　リファクタリング

自分が作成したプログラムは、拡張性がなく、可読性が低い。コマンドとしてほかにも考えられる入力が存在するが、あらゆる場合に対応するとなるとプログラムがより複雑になる。そのため、他人が作成したプログラムを読み、コードの見やすさや拡張性について理解を得た。参考プログラムから、リファクタリングを施した点は次の 3 つである。

(1) コマンドの内容を構造体で管理

(2) リスト構造でデータを柔軟に管理
(3) 逐次処理に変更

次に、リファクタリングを施した 3 つの点を具体的に説明する。

4.1 コマンドの内容を構造体で管理

構造体は、同じ種類のデータを管理することに長けている。リダイレクト、パイプなどのコマンドを実行するには、必要な情報が異なるが、リダイレクトの処理は同じ情報で処理できる。このように、同じタイプのデータをまとめて構造体で管理することで、処理の見通しが良くなると考える。次に実際に用いた構造体の定義を示す。

ソースコード 2　利用した構造体

```
1  struct cmd{ ···(1)
2    int type;
3  };
4
5  struct execcmd { ···(2)
6    int type;
7    char *argv[MAXARGS];
8  };
9
10 struct redircmd { ···(3)
11   int type;
12   struct cmd *cmd;
13   char *file;
14   int mode;
15   int fd;
16 };
17
18 struct pipecmd { ···(4)
19   int type;
20   struct cmd *left;
21   struct cmd *right;
22 };
```

(1) 基底構造体
基底構造体の役割は、クラスの抽象化と似ている。プログラム全体を通して同一の型を持つ構造体を用いることで、様々な処理を統一的に管理することができる。

(2) コマンド実行構造体
コマンド実行構造体は、ls や echo hello のようなコマンドを実行するための構造体である。argv の配列にコマンドの文字列を代入して、execvp 関数で実行する。

(3) リダイレクト構造体
リダイレクト構造体は、コマンドの出力先を標準に入力からファイルに変更するための情報を管理するための構造体である。ここで、struct cmd *cmd は、出力先をファイルに変更されたコマンドである。

(4) パイプ構造体
パイプ構造体は、パイプ処理を行う部分に関する構造体である。パイプ構造体自体はパイプのための処理を行わず、再帰処理を行うために利用される。

4.2 リスト構造でデータを柔軟に管理

リスト構造は、プログラム中にデータの個数が変化するものを扱うときに有効である。また、リスト構造を用いると再帰処理を記述しやすい。例えば、リダイレクトの処理はファイルの出力先を変更することと、コマンド実行する 2 つ行う必要がある。リストでそれぞれの手順を示すことができれば簡潔にコードを記述できる。次に示すコードはリダイレクト部分を記述した runcmd を一部抜粋したものである。

ソースコード 3　runcmd 一部抜粋

```
1  case REDIR:
2    rcmd = (struct redircmd*)cmd;···(1)
3    int fd=open(rcmd->file, rcmd->mode,
           S_IRUSR | S_IWUSR | S_IRGRP |
         S_IROTH);
4    if (fd < 0) {
5      perror("open failed");
6      exit(-1);
7    }
8    dup2(fd,rcmd->fd);
9    close(fd);
10   runcmd(rcmd->cmd); ···(2)
11   break;
```

(1) キャスト

基底構造体である cmd を、特定の構造体にキャストすることでリダイレクト構造体に変更することができる。リダイレクト構造体にキャストされたものは、リダイレクト情報をメンバに持つ。

(2) 再帰処理

この再帰処理は、リダイレクト構造体のメンバである cmd を引数としてもう一度 runcmd を行う。これにより、出力先を変更した後にコマンドを実行することができる

4.3 逐次処理に変更

自分が作成したプログラムは、コマンド一度すべて見てからどのようなコマンドが入力をされたか確認し、コマンドに応じた処理を行う。それに対して、修正したプログラムは先頭から文字列を見ていき、リダイレクトやパイプの場合はどうすればいいかを同一の手順で行うようにする。次に示すコードは parsecmd の一部を抜粋したコードである。

ソースコード 4　parsecmd 一部抜粋

```
exe = execcmd();
cmd = (struct execcmd*)exe;
while(argv[idx]!=NULL){
  if(strchr("<>",argv[idx][0])){
    if(strchr("<>|",argv[idx
        +1][0])){ ···(1)
      printf("syntax error\n");
      exit(-1);
    }
    switch(argv[idx][0]){ ···(2)
      case '<':
        exe=redircmd(exe,argv[idx+1],
            O_RDONLY, 0);
        break;
      case '>':
        if(argv[idx][1]=='>'){
          exe=redircmd(exe,argv[idx
              +1],O_WRONLY|O_CREAT,
              1);
        }else{
          exe=redircmd(exe,argv[idx
              +1],O_WRONLY|O_CREAT,
              1);
        }
        break;
    }

    idx+=2;
  }else if(strcmp(argv[idx],"|"
      )==0){
    idx++;
    exe=pipecmd(exe,parsepipe(argv));
        ···(3)
  } else { // コマンドのとき
    argc=0;
    while(argv[idx]!=NULL){ ···(4)
      if(strchr("<>",argv[idx][0])){
        break;
      }else if(strcmp(argv[idx],"|"
          )==0){
        break;
      }else{
        cmd->argv[argc++]=argv[idx];
        idx++;
      }
    }
    cmd->argv[argc]=NULL;
  }
}
idx=0;
return exe;
```

(1) エラー処理

リダイレクトの記号の次のコマンドブロックに、また同じリダイレクトまたはパイプ記号が存在するとき、コマンド書き方がおかしいとして、syntax error と出力するようにした。

(2) リダイレクトの種類を検出

リダイレクトの記号の種類を判別して、記号に応じて処理を変更する。

(3) パイプの再帰処理

パイプの左のポインタには、パイプのひとつ前のコマンドの構造体のアドレスを格納する。右には、パイプの後のコマンド内容を書き込む

(4) コマンド処理

コマンドを見つけた場合は、リダイレクトやパイプの記号になるまで cmd 構造体の argv にコマンド文字列を代入する。

5　実験

今回の実験は、リファクタリングしたコードは本来通りの動きを実現できていることとともに、コマンドが間違っている場合は正常にコマンド実行されず、syntax error と出力されるか確認する。次に、シェルプログラムを実行したときの実行結果である。ls > >と入力した場合はシェル上に syntax

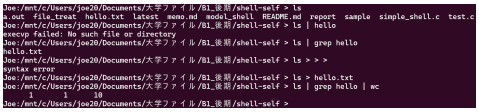

図 1　自作シェル実行結果

error と表示され、コマンドの書き方が間違っていると指摘している。通常のコマンド、パイプの実行、パイプが多段になった場合でも正常にプログラムが動作していることがわかる。また、リダイレクトの処理が正しく行われていることを確認するために、hello.txt の内容を次に示す。

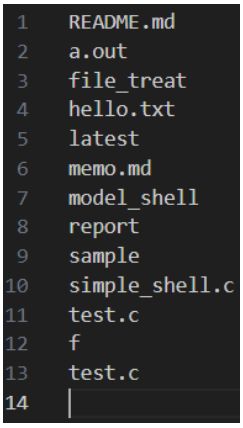

図 2　hello.txt 中身

確かに、ls の出力内容が hello.txt に反映されていることがわかる。

6　まとめ

本レポートでは、シェルの動作原理とリファクタリングについて学ぶことを目的とした。空白で区切られたコマンドの形式が定まっていることで、コマンドを容易に処理することができるのだと理解した。また、リファクタリングを行ったことで、リスト構造をうまく利用して簡潔かつ拡張性に優れたコードを記述する技術を取得できた。

付録

付録 1　パイプリダイレクト関数

```
1  void PipeRedirect(char **argv){
2    int is_redirect=0;
3    int is_oppose_redirect=0;
4    int *pipe_locate=NULL;
5    pipe_locate=(int*)malloc(sizeof(
         int));
6    pipe_locate[0]=-1;
7    int pipe_count=0;
8
9    for(int i=0;argv[i]!=NULL;i++){ ··
         ·(1)
10     if(strcmp(argv[i],">")==0){
11       is_redirect=i;
12       argv[i]=NULL;
13     }else if(strcmp(argv[i],"<"
           )==0){
14       is_oppose_redirect=i;
15       argv[i]=NULL;
16     }else if(strcmp(argv[i],"|"
           )==0){
17       pipe_count++;
18       pipe_locate=realloc(pipe_locate
             ,(pipe_count+1) * sizeof(
             int));
19       pipe_locate[pipe_count]=i;
20       argv[i]=NULL;
21     }
22   }
23
24   int pipefd[pipe_count+1][2];
25
26   if(pipe_count==0 && is_redirect==0
         && is_oppose_redirect==0){ ···
         (2)
27     if(fork()==0){
28       if(execvp(argv[0],argv)){
29         perror("execvp");
30         exit(EXIT_FAILURE);
31       }
32     }
33   }else if(is_oppose_redirect!=0){ ··
         ·(3)
34     if(fork()==0){
35       if(is_FileOrDir(argv[is_redirect
             +1])==0){
36         exit(1);
37       }
38       if(is_FileOrDir(argv[
             is_oppose_redirect+1])==0){
39         exit(1);
40       }
41       int fd_in=open(argv[
             is_oppose_redirect+1],
             O_RDONLY);
42       dup2(fd_in,STDIN_FILENO);
43       close(fd_in);
44       if(is_redirect!=0){
45         int fd_out=open(argv[
               is_redirect+1], O_CREAT |
                O_WRONLY, S_IRUSR |
               S_IWUSR | S_IRGRP |
               S_IROTH);
46         dup2(fd_out,STDOUT_FILENO);
47         close(fd_out);
48       }
49       if(execvp(argv[0],argv)){
50         perror("execvp");
51         exit(EXIT_FAILURE);
52       }
53     }
54   }else if(is_redirect!=0 &&
         pipe_count==0){ ···(4)
55     if(fork()==0){
56       if(is_FileOrDir(argv[is_redirect
             +1])==0){
57         exit(1);
58       }
59       int fd=open(argv[is_redirect
             +1], O_CREAT | O_WRONLY,
             S_IRUSR | S_IWUSR | S_IRGRP
              | S_IROTH);
60       dup2(fd, 1);
61       close(fd);
62       if(execvp(argv[0],argv)){
63         perror("execvp");
64         exit(EXIT_FAILURE);
65       }
66     }
```

```
67  }else{ ···(5)
68    for(int i=0;i<pipe_count+1;i++){
69      if(i!=pipe_count){
70        pipe(pipefd[i]);
71      }
72      if(fork()==0){
73        if(i==0){
74          close(pipefd[i][0]);
75          dup2(pipefd[i][1],
                STDOUT_FILENO);
76          close(pipefd[i][1]);
77        }else if(i==pipe_count){
78          dup2(pipefd[i-1][0],
                STDIN_FILENO);
79          close(pipefd[i-1][0]);
80          if(is_redirect){
81            int fd=open(argv[
                  is_redirect+1],
                  O_CREAT | O_WRONLY,
                  S_IRUSR | S_IWUSR |
                  S_IRGRP | S_IROTH);
82            dup2(fd, 1);
83            close(fd);
84          }
85          close(pipefd[i-1][1]);
86        }else{
87          close(pipefd[i][0]);
88          close(pipefd[i-1][1]);
89          dup2(pipefd[i-1][0],
                STDIN_FILENO);
90          dup2(pipefd[i][1],
                STDOUT_FILENO);
91          close(pipefd[i][1]);
92          close(pipefd[i-1][0]);
93        }
94        if(execvp(argv[pipe_locate[i]
              + 1], argv + pipe_locate[
              i] + 1) < 0){
95          perror("execvp");
96          exit(EXIT_FAILURE);
97        }
98      }else{
99      }
100   }
101 }
102 for(int i=0;i<pipe_count-1;i++){
103     wait(NULL);
104   }
105   free(pipe_locate);
106 }
```

参考文献

[1] エラトステネスのふるい, 2024/12/15, https://algo-method.com/descriptions/64

[2] webpia, 2024/12/15, https://webpia.jp/quick_sort/

[3] Qiita, 2024/12/15, https://qiita.com/cotrpepe/items/1f4c38cc9d3e3a5f5e9c