[Lv4]minishell (3/3)

♡ 11

syamashi
2021年3月21日 10:01

<- 前：minishell(2/3) : コマンドの実行

実装前に。

0. テストツールをつくりましょう

実行結果を得られるようになったら、手入力からテストツールでの確認に切り替えます。デグレード、絶対に頻発します。手入力テストをしないくらいがちょうどいいかもしれません。

※網羅せず、スピード重視で取り組むのも大切かも。

https://github.com/fkymy/minishell-helper

https://github.com/nafuka11/42_minishell_tester

------実装：3 解析------

ゴールは2点。

・execveにchar **argvとchar **envpを渡すこと。

・リダイレクトの出力先を管理。

です。

Q. リダイレクト？

A. 出力先を変更する機能です。

```
# 標準出力が、fileのfdにおきかわる
bash-3.2$ echo aaa > file
bash-3.2$ cat file

aaa
```

各種リダイレクトのオプション設定ですが、straceで確認できました。チートシート。

```
>  : open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
>> : open(path, O_WRONLY | O_CREAT | O_APPEND, 0666);
<  : open(path, O_RDONLY);
```

1. まずlineの文頭文末から、SPACEと/tを削除します。ft_strtrimが便利。

2. 要素ごとに分解します。

Q. 要素には何がある？

A. https://linuxjm.osdn.jp/html/GNU_bash/man1/bash.1.html

```
メタ文字 (metacharacter)
クォートされていない場合に、単語区切りとなる文字。 以下の文字のうちのいずれかです:
| & ; ( ) < > space tab

制御演算子 (control operator)
制御機能を持つ トークン。 以下のシンボルのうちのいずれかです:
|| & && ; ;; ( ) | |& <newline>
```

分解例はこちら。

要素がいくつになるかわからないので、逐次要素を増やせる線形リスト(t_list)が、扱いやすいです。

```
minishell$ echo " double " ' single ' > f1>>f2<f3

0: 文字列
6: SPACE

[echo][ 0]
[ ][ 6]         <- 要素の区切りにSPACEを挿入
["][ 8]
[][ 0]          <- クオートの開始地点に空文字挿入
[ double ][ 0]
["][ 8]
[ ][ 6]
['][ 7]
[][ 0]
[ single ][ 0]
['][ 7]
[ ][ 6]
[>][ 1]
[ ][ 6]
[f1][ 0]
[ ][ 6]        <- 文字がくっついていてもSPACEを挿入
[>>][ 2]
[ ][ 6]
[f2][ 0]
[ ][ 6]
[<][ 3]
[ ][ 6]
[f3][ 0]
[ ][ 6]
[;][13]
```

上手にわけるコツ2点、

・要素ごとにスペースで分ける

・クオートの直後にダミーの空文字を挿入

分解フラグは、以下17個（2個未使用）です。

```
# define STR 0      // 文字列
# define RDIR 1     // >
# define RRDIR 2    // >>
# define LDIR 3     // <
# define LLDIR 4    // <<
# define LLLDIR 5   // <<<
# define SPACE 6    //
# define SQUOTE 7   // '
```

```
# define DQUOTE 8   // "
# define PIPE 9     // |
# define DPIPE 10   // ||
# define AND 11     // &
# define DAND 12    // &&
# define SCOLON 13  // ;
# define DSCOLON 14 // ;;
# define DOLL 15    // $ 使わなかった
# define ESC 16     // \
# define SSTR 17    // single_quote内の文字列。 使わなかった。
# define RINT 18    // 2>file の[2]
```

3. 渡された文字が正しいかsyntax_errorのチェックをします。

次の6要素 + 4要素でもれなくblockできました。

```
---syntax_check---
  2つの群、rd{<, >, >>}、meta {|, ;}、について。次の6要素のどれかならOUT
  1. 先頭 meta OUT
  2. meta meta OUT
  3. rd meta OUT
  4. rd rd OUT
  5. rd EOF OUT
  6. ;; OUT

---avoid check---
  subjectで、しないでいいとされているものをエラー処理する場合
  7. open quote BLOCK  // クオートが開いている
  8. | EOF BLOCK       // 最後がpipe
  9. ESC EOF BLOCK     // 最後がエスケープキー

  10. BONUS metas{&&, ||, >>>} BLOCK // 実装しない記号

minishell: ; ;
minishell: syntax error near unexpected token ';'
minishell$ echo $?
258

minishell$ "echo
minishell: (*д*)oops 'open quote' does not support...
```

4. 適切な文字列とわかったら、さらに解析を進めていきます。

こちらが完成されたものです。

```
minishell$     echo " double " ' single ' > f1>>f2<f3

[echo][ 0]
[ double ][ 0]
[ single ][ 0]
[>][ 1]
[f1][ 0]
[>>][ 2]
[f2][ 0]
[<][ 3]
[f3][ 0]


以下の手順を実装しています。

->文字列内の環境変数を展開
->空文字["", 0]の削除
->クオーテーション[7][8]の削除
->文字列[0]+文字列[0]の結合
-> SPACE[6]の削除
```

5.ここから、[ast 抽象構文木] なる木構造に分解していきます。

木構造でデータをわけたものを、一般的にASTと呼ぶのかも？

（概念の参考にしました。再現してません）

https://dev.to/oyagci/generating-a-parse-tree-from-a-shell-grammar-f1

渡された入力ですが、セミコロン単位で実行がなされます。

セミコロン単位の大枠に分けるため、枝を生やします。

これは「t_listの(void *)contentの中に、t_listを入れる」で実装できます。

```
bash-3.2$ echo a ; echo b; echo c
a
b
c

---semi[0]---
[echo][ 0]
[a][ 0]

---semi[1]---
[echo][ 0]
[b][ 0]
```

```
---semi[2]---
[echo][ 0]
[c][ 0]
```

Q. t_listにt_listってどういれるの？

A. ft_lstnew()の中にlistを入れるといいです。

```
t_list *boss;
t_list *new;
t_list *list;

boss = NULL;
new = ft_lstnew(list);
ft_lstadd_back(&boss, new);

((t_list*)boss->content)->contentとすれば、*listのcontentにたどり着けます。
          --------------
          ここにt_list *listが入っているけど、(void*)なので、(t_list *)でキャストします。

もしlist->content = char *lineだったら？

(char *)(((t_list*)boss->content)->content)
でキャストしてあげればとれます。
```

6. リダイレクトの有効範囲がpipe単位です。

なので、さらにpipe単位でリダイレクトとあわせて分ける必要もあります。

argvとリダイレクト(rd)の葉っぱをはやします。

これは「t_listのcontentの中に、t_listが2つ入った構造体を渡す」で実装できます。

```
bash-3.2$ echo aaa > file < file bbb ccc ; cat file | wc; echo c
      1       1       4   // wcの出力
c                         // echo c


---semi[0]---
pipe[0]
```

```
  argv : echo, aaa, bbb, ccc
  rd : > file < file

---semi[1]---
pipe[0]
  argv : cat, file
  rd :
pipe[1]
  argv : wc
  rd :

---semi[2]---
pipe[0]
  argv : echo, c
  rd :
```

ゴールは、argvが渡せればよいので、semiのリストをlaunch担当に渡すことになります。

Q. ＊が多くて型があわない

A. ＊の優先度は一番最後、と知っておけば怖くないです。

```
int *a

*a = 1;
(*a)++; // これは*aの値を増加
*a++;  // これはaのポインタ位置が１つずれて、その値のポインタをとる。*は最後に処理されます。
```

・各種builtin関数の実装にあたって

bashのソースコードを読むと多くの気づきが得られます。

例えばcd

http://git.savannah.gnu.org/cgit/bash.git/tree/builtins/cd.def

目に留まった条件分岐などをかいつまむだけでも、例えば$PWD、$OLDPWD、$CDPATHなどの変数と絡みがある、ぐらいのことが分かってきます。

echo
http://git.savannah.gnu.org/cgit/bash.git/tree/builtins/echo.def

list->word->wordは先頭文字を見てるかな、とか。それでオプション判定かなぐらいのことが見えてきます。

一言。
条件分岐とても多くなりますが、「情報整理してから実装すべきだった」が、私の一番の収穫でした。

最後に、提出前チェックシート。絶対。

・linuxで valgrind --leak-check=full --show-leak-kinds=all
・ワカモレでnorminette
・ワカモレで-g fsanitize=address
・ワカモレでnm -u minishell