



Nafuka Lines

[読者になる](#)

2021-03-29

bash再実装の課題振り返り

#42 [#bash](#) [#C言語](#)

[42 Tokyo](#) の課題でbashの再実装に取り組みました。本記事ではその振り返りをします。

- [課題の概要](#)
- [成果物](#)
- [取り組み方](#)
 - [調査](#)
 - [開発方法](#)
- [実装](#)
 - [1. Lexer（単語分割）](#)
 - [状態の更新](#)
 - [単語分割](#)
 - [2. Parser（構文解析）](#)
 - [抽象構文木](#)
 - [構文解析方法](#)
 - [エラー処理](#)
 - [3. Expansion（変数展開）](#)
 - [4. Command execution（コマンド実行）](#)
 - [パイプ](#)
 - [パイプの実行方法](#)
 - [リダイレクト](#)
 - [builtinコマンドでの注意点](#)
 - [工夫したこと](#)
 - [データ構造の可視化](#)
 - [テスターの作成](#)
- [感想](#)
 - [楽しかったこと](#)

- 大変だったこと
- 終わりに

課題の概要

機能が制限されたbashを再実装する課題です。

具体的には、

- Simple command
- いくつかのbuiltin command
 - cd, echo など。env など本来builtinでないコマンドも含みます
- 構文解釈 (; | " ')
- リダイレクト (< > >>)
- 環境変数の展開

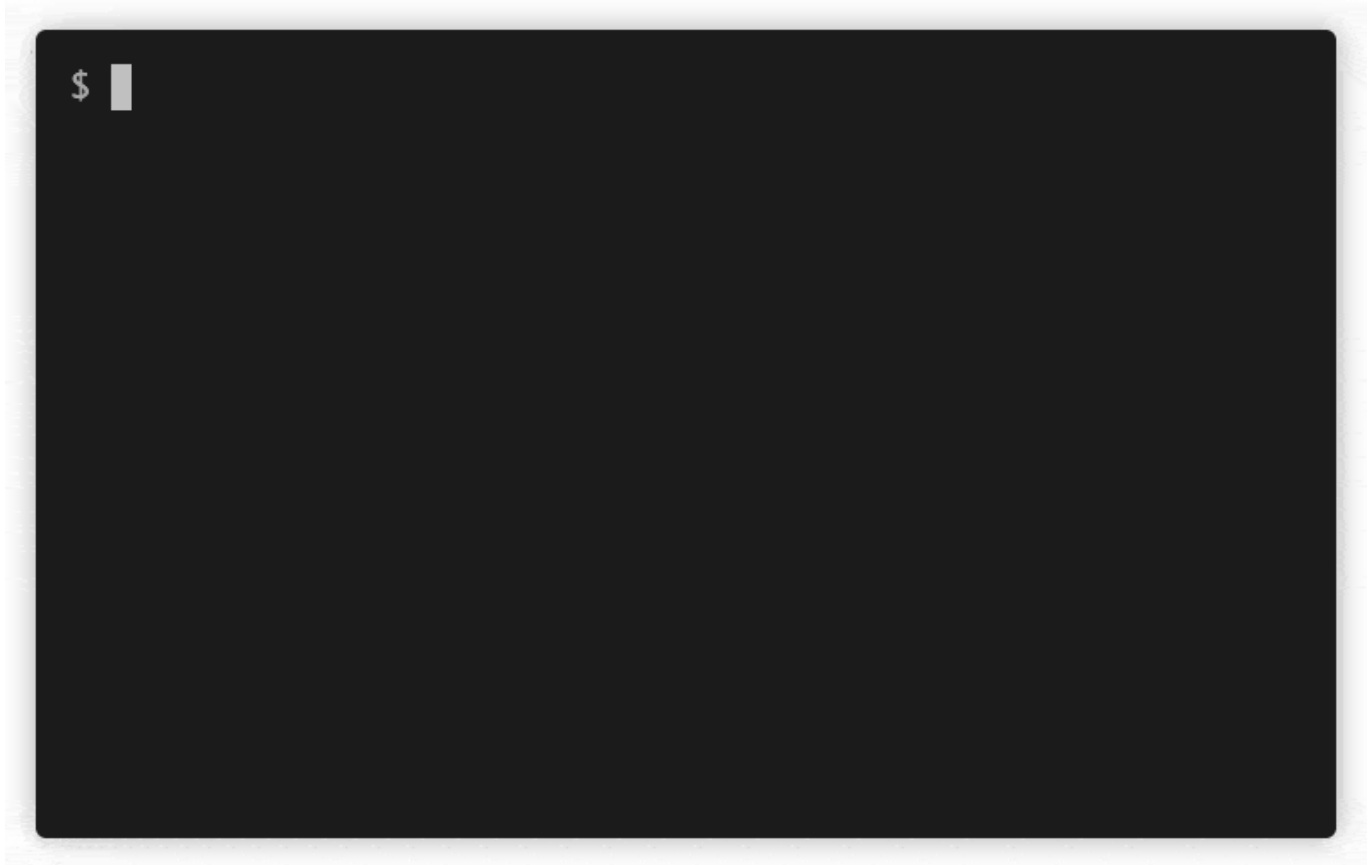
を実装します。

課題は2人の生徒で取り組む必要があります。

この課題をこなすことで、以下のことが学べたと思います。

- bashの仕様
- 構文解析 (LexerとParser)
- プロセスの作成、任意のプログラム実行
- パイプ、リダイレクション
- シグナルハンドリング

成果物



GitHub - ToYeah/42_minishell: A small shell like bash.

A small shell like bash. Contribute to ToYeah/42_minishell development by creating an account on GitHub.



h/
nishell

« bash.

0 Issues 3 Stars 2 Forks

github.com

取り組み方

42 Tokyoの生徒、[ToYeah](#)さんと課題に取り組みました。

調査

まずは `bash` で何を実装すべきか調べました。

[The Architecture of Open Source Applications](#) というオープンソースアプリケーションの設計について書かれた本の [日本語訳](#) を読みました。

この本には`bash`のアーキテクチャについて載っていました。

本を読んだところ、bashのメイン部分の実装は以下の4工程に分けられそうでした。

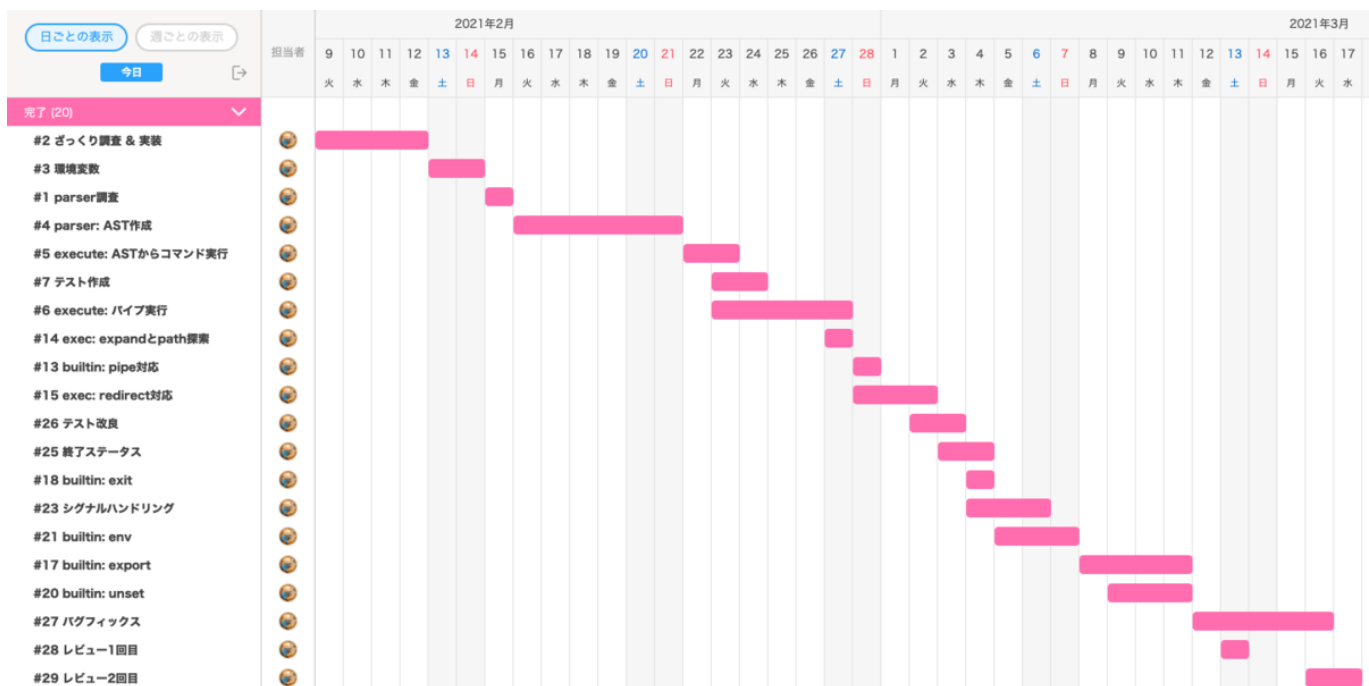
1. Lexer (単語分割)
2. Parser (構文解析)
3. Expansion (変数展開)
4. Command execution (コマンド実行)

開発方法

開発は GitHub 上で行いました。

コードに変更を加える場合は基本的に Issue を立てた後、Issue に紐づく PullRequest を出し、相手にレビューしていただきました。

プロジェクト管理は [Jooto](#) でガントチャートを作り、進捗を確認していました。
自分の進捗はこんな感じでした。課題着手から完了まで、おおよそ37日かかったようです。



ドキュメントは [HackMD](#) を使いました。議事録を残したり、参考URLを共有していました。

実装

1. Lexer (単語分割)

Lexerでは入力文字列を構文解析しやすいよう、単語に分割します。

例えば、

```
echo "hello w"'orld'; cat<file|wc
```

という入力文字列が渡された場合、

```
echo
"hello w"'orld'
;
cat
<
file
|
wc
```

に分割します。

状態の更新

入力文字列を先頭から1文字ずつ見ていって、状態を更新していきます。

状態は以下の3つのいずれかです。

1. 通常
2. シングルクォートの中
3. ダブルクォートの中

今見ている文字が `'` なら状態をシングルクォートの中に、 `"` なら状態をダブルクォートの中に更新します。

対応するクォートが現れたら、状態を通常に戻します。

単語分割

状態が通常の際にスペース、 `;` `|` `>` など、単語を分割すべき文字を見つけたら、そこで単語を分割します。

Lexerの段階では、まだ `"` `'` は残ったままです。 `"` `'` は 3. Expansion で除去します。

2. Parser（構文解析）

Parserでは、分割された単語を先頭から1つずつ見て構文解析し、抽象構文木を作ります。

抽象構文木

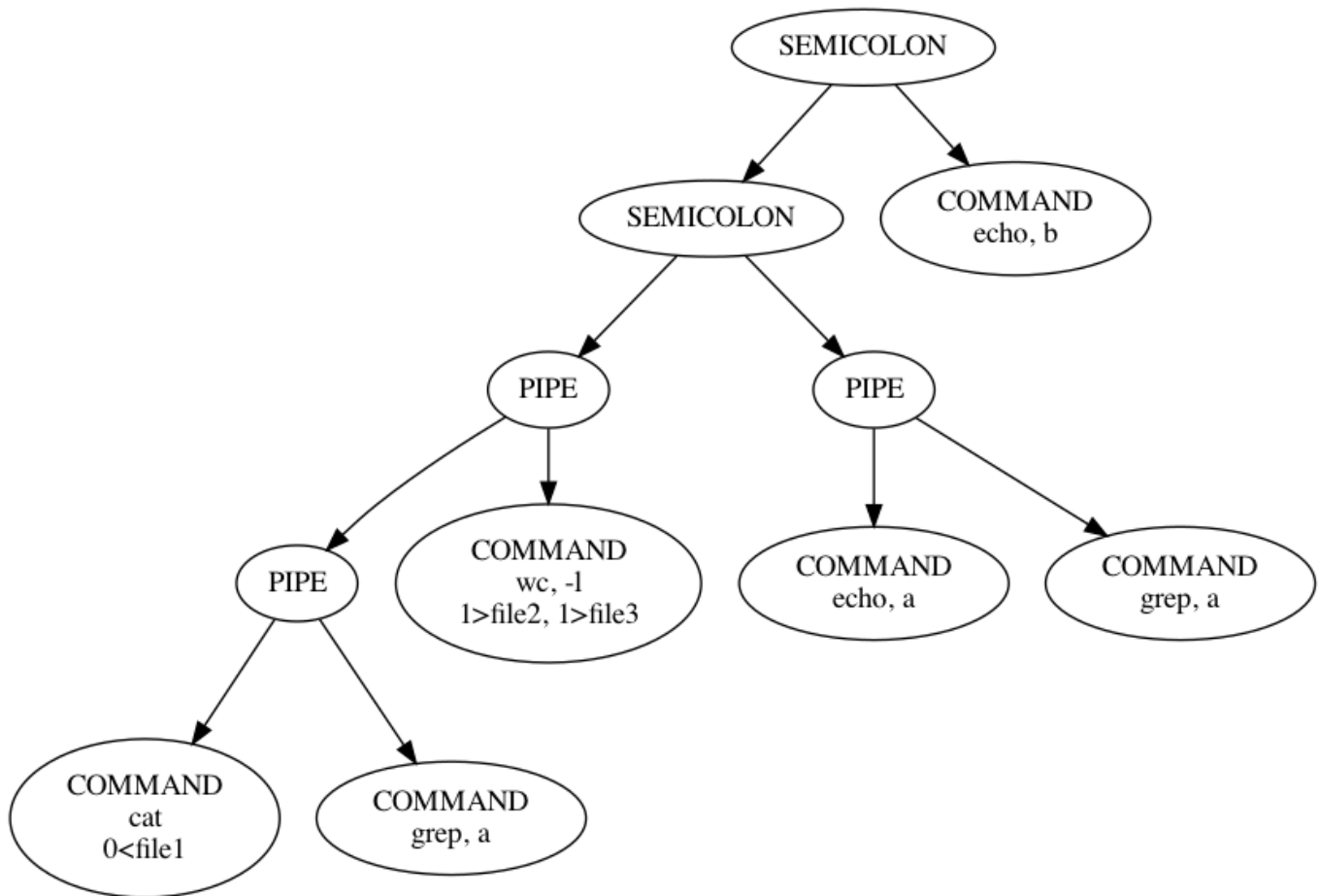
抽象構文木とは、構文解析した結果をプログラムで処理しやすいよう木構造にしたデータです。

構文解析はリストでも実現できるようですが、木の方が文法の追加に柔軟に対応できそうなのでこちらを採用しました。

私たちの実装では、この入力文字列は、

```
cat < file1 | grep a | wc -l > file2 > file3 ; echo a | grep a; echo b
```

以下の図のようなデータ構造になります。



木の左下から右上にたどると、入力文字列と対応するようになっています。
コマンドノードには引数のリスト、リダイレクトのリストを持つようにしています。

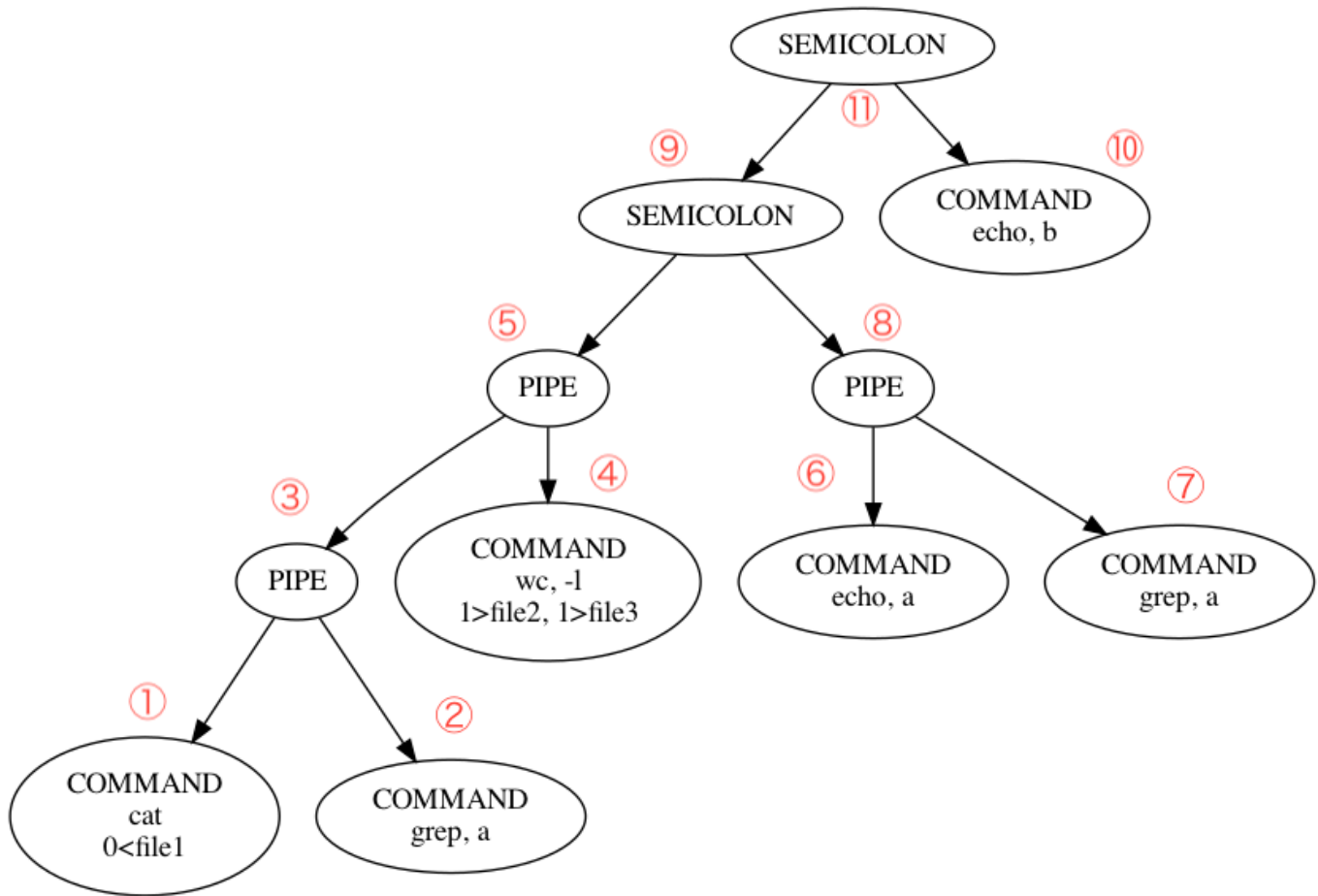
構文解析方法

シェルの文法に関しては [POSIXの規格](#) に載っている「Shell Grammar Rules」を参考にしました。[バックス・ナウア記法](#)を理解するために、[Railroad Diagram Generator](#) で可視化して文法を確認しました。

規格を見たところ、今回の課題では `;` → `|` → `command` の順に解釈していけば良さそうでした。

- `;` の解釈では、`|` の解釈の後、`;` の解釈と `|` の解釈のループに入ります。`;` がなくなったらループを抜けます。
- `|` の解釈では、`command` の解釈の後、`|` の解釈と `command` の解釈のループに入ります。`|` がなくなったらループを抜けます。
- `command` の解釈では、リダイレクト (`<` `>` `>>`) が来たら次の単語をファイル名とし、それ以外はコマンド文字列とします。`|` や `;` が来たり、単語がなくなったらループを抜けます。

.....と文章で書いても難しいかもしれません。先ほどの抽象構文木の図に番号を振ってみました。



番号順に木が作られます。左下の末端からコマンドの子ノードを作り、親ノードに繋げていきます。

構文解析の方法は以下のURLを参考に作成しました。

- [低レイヤを知りたい人のためのCコンパイラ作成入門](#)
 - [文法の記述方法と再帰下降構文解析](#) が分かりやすかったです。
- [Swoorup/mysh: A basic unix shell interpreter in c programming language using recursive descent parser.](#)
 - 再帰下降構文解析を使ったbashの実装例です。

エラー処理

`;` の前にコマンドがない場合、`|` や `>` の後に単語がない場合は構文解析エラーになります。

`echo >` のような入力は構文解析エラーになります。一方で `> file;` は許容されます。

3. Expansion (変数展開)

Expansionは、コマンド実行前に呼び出され、コマンドの引数とリダイレクトの単語を処理します。

単語に含まれる `$USER` のような変数を展開し、`"` `'` の除去も行います。

変数展開によって単語が分割されるため、ここで再度Lexerを実行します。

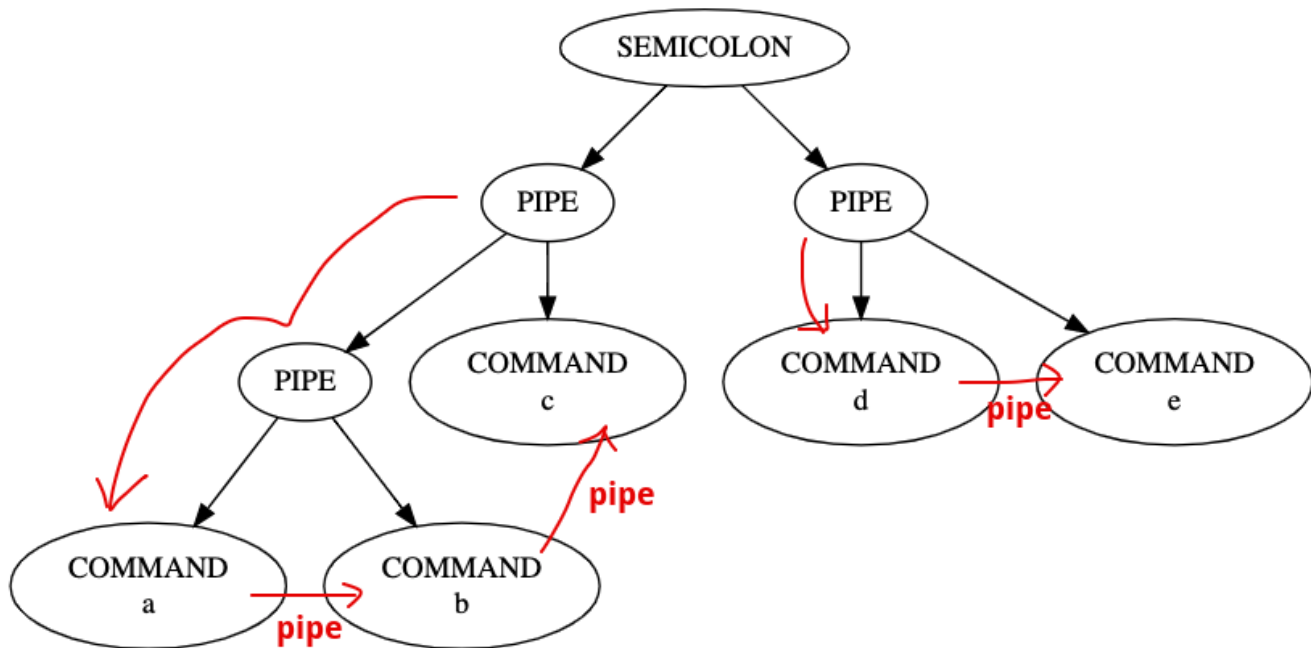
例えば、`VAR="echo hello"` という変数があり、`$VAR` という1つの単語が与えられたら、2つの単語 `echo` , `hello` に分割されます。

4. Command execution (コマンド実行)

抽象構文木の左下まで降りて、右上に向かってコマンドを実行していきます。

パイプ

パイプするためには、木を登って降りてコマンドノードを見ていく必要があります。今回は木を登らず、構文解析時にコマンドノードをポインタで繋ぐことにしました。



パイプの実行方法

パイプが1つの場合

`a | b` のような場合は、1つパイプを作って、それぞれのプロセスの標準入出力を繋ぎます。

パイプが複数の場合

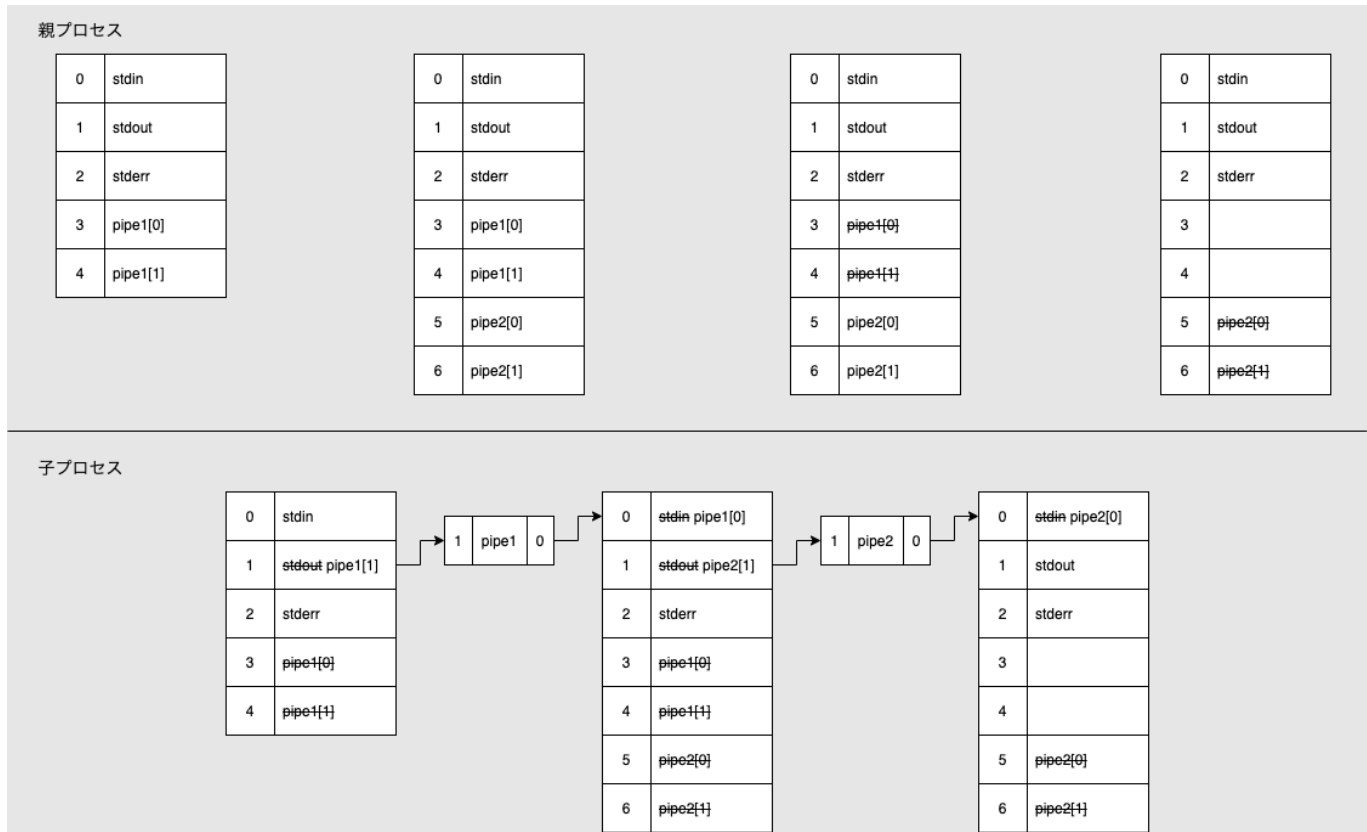
`a | b | c` のようにパイプが複数になった場合、少し複雑になります。コマンド `b` を実行する時には 入力側のパイプ、出力側のパイプ、2つのパイプが必要です。

これはパイプの最初、中間、最後と3つの状態で場合分けして考えられます。
パイプが増えると中間状態のコマンドが増えます。

1. パイプの最初
- パイプを作成し、パイプの片側を閉じ、片側を標準出力に繋がります。
2. パイプの中間
- 1で作成したパイプの片側を閉じ、片側を標準入力に繋がります。
 - パイプを作成し、パイプの片側を閉じ、片側を標準出力を繋がります。
3. パイプの最後
- 2で作成したパイプの片側を閉じ、片側を標準入力に繋がります。

私たちは `pipe_state` というenumで状態を持たせました。bitで read, writeのフラグを管理しても良かったかもしれません。

パイプの処理を図にするとこんな感じになります。



注意点として、使い終わったパイプは親、子プロセスともに閉じないといけません。そうしないと、パイプがずっと入力を待ち続けてしまいます。

リダイレクト

リダイレクトはパイプの後に行います。パイプする前にファイルを開いておき、パイプ後にファイルディスクリプタを上書きします。

builtinコマンドでの注意点

パイプで繋がれていないbuiltinコマンドの場合、子プロセスでなく親プロセスで実行されます。子プロセスにしてしまうとexportなど環境変数の設定が親プロセスに反映されないためです。

親プロセスでリダイレクトすると、親の標準入力、標準出力などが書き換わり、その後のコマンド実行に影響が出ます。

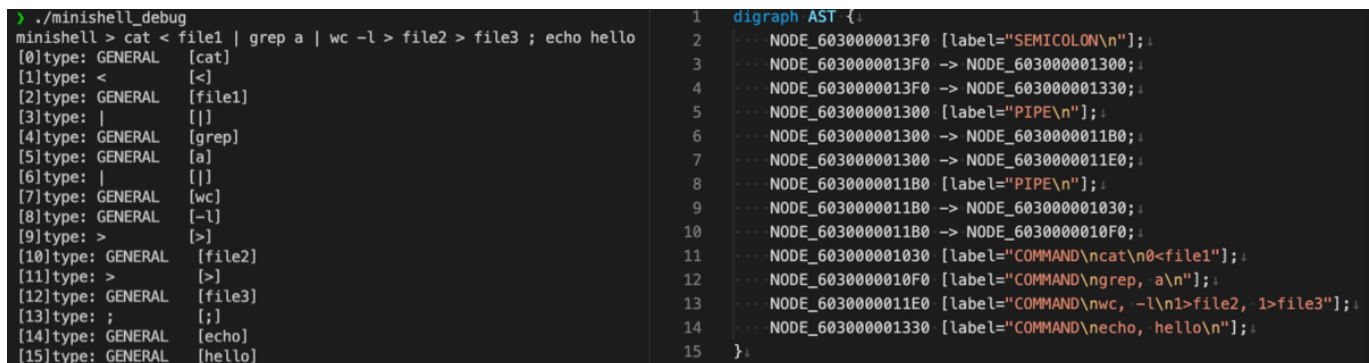
このため、上書きする前にファイルディスクリプタを `dup` してバックアップし、リダイレクトが完了したら `dup2` して復元する必要があります。

工夫したこと

データ構造の可視化

デバッグ時に、抽象構文木をdotファイルに出力、単語リストを標準出力するようにしました。

画像の左側が単語リスト、右側がdotファイルです。



```
./minishell_debug
minishell > cat < file1 | grep a | wc -l > file2 > file3; echo hello
[0]type: GENERAL [cat]
[1]type: < [<]
[2]type: GENERAL [file1]
[3]type: | [|]
[4]type: GENERAL [grep]
[5]type: GENERAL [a]
[6]type: | [|]
[7]type: GENERAL [wc]
[8]type: GENERAL [-l]
[9]type: > [>]
[10]type: GENERAL [file2]
[11]type: > [>]
[12]type: GENERAL [file3]
[13]type: ; [;]
[14]type: GENERAL [echo]
[15]type: GENERAL [hello]

1 digraph AST {
2     NODE_6030000013F0 [label="SEMICOLON\n"];
3     NODE_6030000013F0 --> NODE_603000001300;
4     NODE_6030000013F0 --> NODE_603000001330;
5     NODE_603000001300 [label="PIPE\n"];
6     NODE_603000001300 --> NODE_6030000011B0;
7     NODE_603000001300 --> NODE_6030000011E0;
8     NODE_6030000011B0 [label="PIPE\n"];
9     NODE_6030000011B0 --> NODE_603000001030;
10    NODE_6030000011B0 --> NODE_6030000010F0;
11    NODE_603000001030 [label="COMMAND\ncat\n<file1\n"];
12    NODE_6030000010F0 [label="COMMAND\ngrep, a\n"];
13    NODE_6030000011E0 [label="COMMAND\nwc, -l\n1>file2, 1>file3\n"];
14    NODE_603000001330 [label="COMMAND\necho, hello\n"];
15 }
```

dotファイルは [Graphviz](#) などのツールを使うと画像を生成できます。本記事の抽象構文木の図はdotファイルを Graphviz で生成したものです。

可視化により、不具合がどこで起こっているのか突き止めやすくなったと思います。また、図示すると説明もしやすかったです。

｜ テスターの作成

早い段階から テスター を作り、リグレッションが起きていないか、メモリリークが起きていないか確認するようにしました。

このおかげで比較的少ない不具合数（cファイル, ヘッダファイルの行数およそ4300行に対し、不具合数23）で済んだように思います。

感想

楽しかったこと

チーム開発の楽しさ

一人と違って、相手から反応があり楽しかったです。相手からどんなレビューが来るか、コードが来るか、楽しみにしていました。

チームメイトの ToYeah さんは実装力があり、レビューでは特にメモリリークについて指摘いただき、頼もしかったです。

構文解析の面白さ

資料を参考に実装したら、こんなに綺麗に作れるのかと感動しました。

大変だったこと

bashの仕様の把握

特にbuiltinコマンドはソースコードを読まないと分からない挙動が多かったです。bashは長い歴史があり、ソースコードも歴史を感じさせるものとなっています（そんなに読みやすくないという意味です）。

以下はbashの面白い挙動の一例です。

- `cd`は、カレントディレクトリが存在しない場合、`cd`するディレクトリをPWDにjoinして`cd`を試みます。
 - この挙動については ToYeah さんが詳しく解説されています。

[bash] カレントディレクトリが存在しなくても `cd ./`は成功する - Qiita

環境 \$ bash --version bash --version GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu) 事象 \$ mkdir test; ...

h] カレントディレクトリ
なくても`cd ./`は成功する

ah0102



qiita.com

- `exit`は、複数の引数が与えられた場合、`exit`しません。そして `;` 以降のコマンドは無視されます。
 - 例えば、`exit 0 0; echo hello` というコマンドの場合、`echo hello` は実行されません。
- シェルがどのくらいの深さで実行されているか確認できる変数、`SHLVL` は1000以上になると `bash: warning: shell level (1000) too high, resetting to 1` と表示され、1にリセットされます。
 - ちなみにbash4.4以前では、`SHLVL=999` の時に`bash`を起動すると `SHLVL` が空文字になるバグがあります。

終わりに

課題を見て、始めはどう実装すればいいか全く分からなかったのですが、資料を読んで協力してなんとか実装できました。

`bash`の仕様や構文解析、プログラムの実行など学ぶことが多く、面白かったです。

時間があれば、コマンドのHistory機能など実装してみたいと思います。

nafuka11 1年前



3

0

ツイート

シェアする

関連記事



2022-07-02

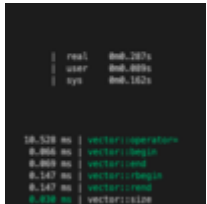
2022年6月振り返り

今年6月に行ったことを振り返る記事です。 エンジニア養成機関...

2021-12-31

2021年振り返り

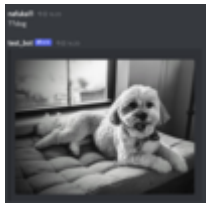
この記事では今年やったことの振り返りをします。 やったこと 4...



2021-12-05

42Tokyoでの2021年11月振り返り

エンジニア養成機関、42Tokyo での活動について知ってもらた...



2021-10-27

Discord botで犬の画像を送信するコマンドを作成しました

はじめに 先日、Discordサーバのユーザから、自作のDiscord bot...



2021-10-02

42Tokyoでの2021年9月振り返り

この記事は、エンジニア養成機関 42Tokyo の活動を知ってもらう...

コメントを書く

[« 2021年2・3月の振り返り](#)

[2021年1月の振り返り »](#)

プロフィール



[nafuka11](#)

読者になる

15

[このブログについて](#)

リンク

[GitHub](#)

[Website](#)

最新記事

[2022年7月振り返り](#)

[2022年6月振り返り](#)

[2022年5月振り返り](#)

[2022年4月振り返り](#)

[C++でHTTPサーバを作った話](#)

カテゴリー

[42 \(30\)](#)

[振り返り \(23\)](#)

[読書感想 \(5\)](#)

[イベント \(5\)](#)

[C++ \(4\)](#)

[Python \(4\)](#)

[C言語 \(2\)](#)

[Discord bot \(2\)](#)

[Hugo \(1\)](#)

[その他 \(1\)](#)[VRChat \(1\)](#)[Serverless Framework \(1\)](#)[Processing \(1\)](#)[GitHub Pages \(1\)](#)[CTF \(1\)](#)[CodinGame \(1\)](#)[bash \(1\)](#)[AWS \(1\)](#)

月別アーカイブ

[▶ 2022 \(9\)](#)[▼ 2021 \(21\)](#)[2021 / 12 \(5\)](#)[2021 / 11 \(2\)](#)[2021 / 10 \(4\)](#)[2021 / 9 \(1\)](#)[2021 / 8 \(2\)](#)[2021 / 7 \(1\)](#)[2021 / 6 \(1\)](#)[2021 / 5 \(1\)](#)[2021 / 4 \(1\)](#)[2021 / 3 \(1\)](#)[2021 / 2 \(2\)](#)[▶ 2020 \(7\)](#)[▶ 2019 \(9\)](#)[▶ 2018 \(2\)](#)

検索

記事を検索

はてなブログをはじめよう！

nafuka11さんは、はてなブログを使っています。あなたもはてなブログをはじめてみませんか？

はてなブログをはじめる（無料）

[はてなブログとは](#)

 [Nafuka Lines](#)

Powered by [Hatena Blog](#) | [ブログを報告する](#)