

# 卒業論文

command line による editor 操作の習熟プログラム

関西学院大学理工学部

情報科学科 西谷研究室

27014533

2018 年 3 月

# 目次

<b>第1章</b>	<b>はじめに</b>	<b>3</b>
1.1	研究の目的 . . . . .	3
1.2	研究の動機 . . . . .	3
<b>第2章</b>	<b>基本的事項</b>	<b>4</b>
2.1	Emacs . . . . .	4
2.2	Ruby . . . . .	5
2.3	RubyGems . . . . .	5
2.4	Keybind . . . . .	6
2.5	CUI(Character User Interface) . . . . .	7
2.6	使用した gem ファイル . . . . .	8
2.6.1	diff-lcs . . . . .	8
2.6.2	Thor . . . . .	8
2.6.3	Minitest . . . . .	9
2.6.4	FileUtils . . . . .	9
2.6.5	Rubocop . . . . .	9
<b>第3章</b>	<b>editor_learner の概要</b>	<b>10</b>
3.1	Installation . . . . .	10
3.1.1	github による install . . . . .	10
3.1.2	gem による install . . . . .	10
3.2	uninstall . . . . .	10
3.2.1	github から install した場合の uninstall 方法 . . . . .	10
3.2.2	gem から install した場合の uninstall 方法 . . . . .	11
3.3	動作環境 . . . . .	11

3.3.1	error 時の対処法 . . . . .	11
3.4	初期設定 . . . . .	12
3.5	delete . . . . .	14
3.6	random_h.rb と sequential_h.rb . . . . .	14
3.7	random_check の動作 . . . . .	17
3.8	sequential_check の動作 . . . . .	18
<b>第 4 章</b>	<b>実装コードの解説</b>	<b>20</b>
4.1	起動時に動作するプログラム . . . . .	20
4.1.1	プログラム内のインスタンス変数の概要 . . . . .	22
4.1.2	File の作成 . . . . .	23
4.2	ファイル削除処理 delete . . . . .	24
4.3	random_check . . . . .	24
4.4	sequential_check . . . . .	27
4.4.1	インスタンス定数に格納されたパス . . . . .	29
4.4.2	動作部分 . . . . .	30
4.5	新しいターミナルを開く open_terminal . . . . .	30
<b>第 5 章</b>	<b>他のソフトとの比較</b>	<b>31</b>
5.1	考察 . . . . .	32
<b>第 6 章</b>	<b>総括</b>	<b>33</b>
<b>第 7 章</b>	<b>謝辞</b>	<b>34</b>

# 図 目 次

2.1	カーソル移動, . . . . .	6
2.2	文字操作, . . . . .	7
3.1	作られるファイルの説明, . . . . .	13
3.2	random_h.rb, . . . . .	15
3.3	全ての操作を終えたターミナル画面, . . . . .	16
3.4	sequential_h.rb, . . . . .	16
3.5	間違っている図, . . . . .	17
3.6	間違っている箇所の表示, . . . . .	18

# 第1章 はじめに

## 1.1 研究の目的

editor\_learner の開発の大きな目的は editor(Emacs) 操作, CUI 操作 (キーバインドなど), Ruby 言語の習熟とタイピング速度, 正確性の向上である. editor 上で動かすためファイルの開閉, 保存, 画面分割といった CUI 操作に習熟することができ, Ruby 言語のプログラムを写経することで Ruby 言語の習熟へと繋げる. 更にコードを打つことで正しい運指を身につけタイピング速度の向上も図っている. コードを打つ際にキーバインドを利用することでキーボードから手を離すことなくカーソル移動などのコマンドを GUI ではなく CUI 操作で行うことにより作業の効率化にも力を入れている. これら全てはプログラマにとって作業を効率化させるだけでなく, プログラマとしての質の向上につながる.

## 1.2 研究の動機

初めはタッチタイピングを習得した経験を活かして, 西谷によって開発された shunkun-type(ターミナル上で実行するタイピングソフト) の再開発をテーマにしていたが, これ以上タイピングに特化したソフトを開発しても同じようなものが Web 上に大量に転がっており, そのようなものをいくつも開発しても意味がなく, それ以外の付加価値を付けたソフトを開発しようと考えた. 西谷研究室ではタイピング, Ruby 言語, Emacs による editor 操作, CUI 操作の習熟が作業効率に非常に大きな影響を与えるので習熟を勧めている. そこでこれらの習熟を目的としたソフトを開発しようと考えた.

## 第2章 基本的事項

### 2.1 Emacs

本研究において使用している editor は Emacs である。Emacs は西谷研究室で使用を勧めている editor であり、非常に強力な editor となっている。

ツールはプログラマ自身の手の延長である。これは他のどのようなソフトウェアツールよりも Editor に対して当てはまる。テキストはプログラミングにおける最も基本的な生素材なので、できる限り簡単に操作できる必要がある [1]。

と書かれている。よって一つの強力な editor を習熟することが勧められている。西谷研究室で勧められている Emacs の機能については以下の通りである、

**設定可能である** フォント、色、ウィンドウサイズ、キーバインドを含めた全ての外見が好みに応じて設定できるようになっていること。通常の操作がキーストロークだけで行えると、手をキーボードから離す必要がなくなり、結果的にマウスやメニュー駆動型のコマンドよりも効率的に操作できるようになります

**拡張性がある** 新しいプログラミング言語が出てきただけで、使い物にならなくなるようなエディタではなく、どんな新しい言語やテキスト形式が出てきたとしても、その言語の意味合いを「教え込む」ことが可能です

**プログラム可能であること** 込み入った複数の手順を実行できるよう、Editor はプログラム可能であることが必須である。

これらの機能は本来エディタが持つべき基本的な機能である。これらに加えて Emacs は、

**構文のハイライト** Ruby の構文にハイライトを入れたい場合はファイル名の後に .rb と入れることで Ruby モードに切り替わり構文にハイライトを入れることが可能になる。

**自動インデント** テキストを編集する際、改行時に自動的にスペースやタブなどを入力しインデント調整を行ってくれる。

などのプログラミング言語に特化した特徴を備えています。強力な editor を習熟することは生産性を高めることに他ならない。カーソルの移動にしても、1 回のキー入力ですべて単位、行単位、ブロック単位、関数単位でカーソルを移動させることができれば、一文字ずつ、あるいは一行ずつ繰り返してキー入力を行う場合とは効率が大きく変わってきます。Emacs はこれらの全ての機能を孕んでいて editor として非常に優秀である。よって本研究は Emacs をベースとして研究を進める。

## 2.2 Ruby

本研究の開発言語として Ruby を使用している。理由は強力な標準ライブラリなどを持っており、構文も自由度が高く記述量も少ない。よって、縛りが少ないのでとっかかりやすくプログラミング言語の中で習熟しやすいと考えたからである。Ruby の基本的な説明は以下の通り、

オープンソースの動的なプログラミング言語で、シンプルさと高い生産性を備えている。エレガントな文法を持ち、自然に読み書きができる [2]。

## 2.3 RubyGems

本研究のソフトは Ruby の gem と呼ばれるパッケージ管理ツールにライブラリとしてリリースしている。さらに gem を利用してファイルの操作やパスの受け渡しなどを行うパッケージを導入している。RubyGems についての説明は以下の通り、

RubyGems は、Ruby 言語用のパッケージ管理システムであり、Ruby のプログラムと ("gem" と呼ばれる) ライブラリの配布用標準フォーマットを提供している。gem を容易に管理でき、gem を配布するサーバの機能を持つ。[3]

Ruby 言語で書かれたプログラムをより便利にするためのツール (ライブラリ) が簡単に使えるように配布されている場所、機能である。本研究で開発したソフトも gem に公開し

である。これにより非常に簡単に本研究で開発したソフトを install できるようになっている。

## 2.4 Keybind

本研究では Keybind の習熟が CUI 操作への適応の第一歩となっており、カーソル移動においても GUI ベースでマウスを使い行の先頭をクリックするより CUI で Control+a と入力して移動する方が高速かつ効率的である。Keybind についての説明は以下の通り、

押下するキー (単独キーまたは複数キーの組み合わせ) と、実行される機能との対応関係のことである。また、キーを押下したときに実行させる機能を割り当てる行為のことである。 [4]

以下 control を押しながらを c- と記述する。習熟するのであれば、どちらの方が早いかは一目瞭然である。本研究は Keybind の習熟による CUI 操作の適応で作業の効率化、高速化に重点を置いている。よく使用されるキーバインドは以下の通り。



Control+f	# 右に移動
Control+b	# 左に移動
Meta+f	# 右に単語単位移動
Meta+b	# 左に単語単位移動
Control+e	# 行末へ移動
Control+a	# 行頭に移動
Control+n	# 下
Control+p	# 上

図 2.1: カーソル移動,



```
Control+h      # 左の文字を削除
Control+d      # 右の文字を削除。何も入力がない状態で使うと現在のシェルのログアウト
Control+k      # カーソルから行末までの文字を切り取り
Control+w      # 左の単語を切り取り
Meta+d         # 右の単語を切り取り
Control+u      # その行全部切り取り
Control+y      # ペースト
Control+/      # undo
```

図 2.2: 文字操作,

## 2.5 CUI(Character User Interface)

本研究は GUI ではなく CUI ベースで開発されている。理由は Keybind と同じで作業の高速、効率化を図るためである。CUI についての基本的な説明は以下の通り、

コンピュータにおいて、キーボード入力と文字表示のみを用いた、ソフトウェアの操作体系。キーボードのコマンド名を入力して操作する方法など。[5]

CUI と GUI にはそれぞれ大きな違いがある。GUI の利点は以下の通り、

- 文字だけでなくアイコンなどの絵も表示できる。
- 対象物が明確な点や、マウスで比較的簡単に操作できる。
- 即座に操作結果が反映される。

CUI の利点は以下の通り、

- コマンドを憶えていれば複雑な処理が簡単に行える。
- キーボードから手を離すことなく作業の高速化、効率化が行える。

今回 GUI ではなく CUI 操作の習熟を目的にした理由は、

- コマンドを憶えることで作業効率が上がる。
- editor 操作の習熟も孕んでいるから。

カーソル移動においても GUI ではなく CUI 操作により、ワンコマンドで動かした方が効率的である。プログラマにとって editor を使わないことなどない。上記の理由から、GUI ではなく CUI 操作の習熟を目的としている。

## 2.6 使用した gem ファイル

### 2.6.1 diff-lcs

diff-lcs は、二つのファイルの差分を求めて出力してくれる。これにより random.check や sequential.check などの正誤判定で誤っていた場合に間違った箇所を表示してくれる。テキストの差分を取得するメソッドは、Diff::LCS.sdiff と Diff::LCS.diff の 2 つがある。複数行の文字列を比較した場合の 2 つのメソッドの違いは以下のとおり。

**Diff::LCS.diff** 比較結果を一文字ずつ表示する。

**Diff::LCS.diff** 比較した結果、違いがあった表について、違いがあった箇所にのみ表示する。

今回使用したのは後者 (Diff::LCS.diff) である。コード自体が長いので全ての比較結果を表示してしまうものすごく長くなってしまう。よって、違いがあった行のみの表示となっている。

### 2.6.2 Thor

コマンドライン作成ツールである Thor によりサブコマンドを自然言語に近い形で設定することができる。Thor についての基本的な説明は以下の通り、

コマンドラインツールの作成を支援するライブラリです。git や bundler のようなサブコマンドツールを簡単に作成することができます。 [6]

Thor の使用でサブコマンドを自然言語に近い形で設定できるので、非常にコマンドが直感的で覚えやすくなっている。

### 2.6.3 Minitest

本研究でソースコードとして利用した本では Minitest が使われている。assert\_equal により出力結果が正しいか判定してくれる。Ruby にはいくつかのテストフレームワークがありますが、Minitest は特別なセットアップが不要、学習コストが比較的低い、Rails 開発に知識を活かしやすい。これらの理由により Minitest を使用している。

### 2.6.4 FileUtils

ファイルのコピーや比較、ファイルの作成などファイル操作については全て FileUtils を利用した。使用した FileUtils は以下の通り、

1. File.join: ファイルのパスとパスを結合する
2. File.exist: ファイルが存在するかどうかの判定。ファイルが存在して入れば”true”存在していなければ”false”を返り値としている。
3. FileUtils.mkdir\_p: ディレクトリ dir とその親ディレクトリを全て作成する。
4. FileUtils.touch: パスに設定されているファイルを作成する。
5. FileUtils.cp: ファイルのコピーを行う。
6. File.expand\_path: パスを絶対パスに展開した文字列を返す。
7. FileUtils.compare\_file: 二つのファイルの比較を行う。一致して入れば”true”一致していなければ”false”を返り値としている。

### 2.6.5 Rubocop

自分の打ち込んだコードがコーディング規則に従っているかをチェックするのに Rubocop と呼ばれる gem を使用した。Rubocop は、

プロジェクトの ruby コードが「コーディング規約どおりに書かれているか」をチェックする静的コード解析ツールである [7]。

自分のコードの体裁を確認するために導入した。

## 第3章 editor\_learnerの概要

### 3.1 Installation

#### 3.1.1 github による install

github によるインストール方法は以下の通りである.

1. "https://github.com/souki1103/editor\_learner" へアクセス
2. Clone or download を押下, SSH の URL をコピー
3. コマンドラインにて git clone(コピーした URL) を行う

上記の手順で開発したファイルがそのまま自分のディレクトリにインストールされる.

#### 3.1.2 gem による install

gem によるインストール方法は以下の通りである.

1. コマンドラインにて gem install editor\_learner と入力, 実行
2. ファイルがホームディレクトの .rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems に editor\_learner が収納される

これで editor\_learner とコマンドラインで入力することで実行可能となる.

### 3.2 uninstall

#### 3.2.1 github から install した場合の uninstall 方法

github から install した場合の uninstall 方法は以下の通りである.

1. ホームディレクトで

1. `rm -rf editor_learner` を入力

2. ホームディレクトリから `editor_learner` が削除されていることを確認する.

以上が `uninstall` 方法である.

### 3.2.2 gem から install した場合の `uninstall` 方法

`gem` から `install` した場合の `uninstall` 方法は以下の通りである.

1. ターミナル上のコマンドラインで

1. `gem uninstall editor_learner` を入力

2. ホームディレクトの `.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems` に `editor_learner` が削除されていることを確認する.

以上が `uninstall` 方法である.

## 3.3 動作環境

Ruby の version が 2.4.0 以上でなければ動かない. 理由としては, `gem` に格納されているパスを正しく受け渡しできないからである. 2.4.0 以下で動作させるためには `editor_learner` の最新 version のみを入れることによって動作することが確認できている. さらに, `open_terminal` が MacOSX でのみ動作するので MacOSX を使用すること.

### 3.3.1 error 時の対処法

error が出た場合は以下の方法を試してください

1. `rm -rf editor_learner` をコマンドラインで入力

これによりファイル生成によるバグを解消できる. もう一つの方法は

1. `gem uninstall editor_learner` をコマンドラインで入力
2. 全ての version を `uninstall` する.
3. 再度 `gem install editor_learner` で最新 version のみを `install` する.

上記の手順により Ruby の version によるバグが解消されることが確認できている. 現在起こるであろうと予想されるバグの解消法は上記の 2 つである. Ruby の version が 2.4.0 以上であればなんの不具合もなく動作することが確認できている.

### 3.4 初期設定

特別な初期設定はほとんどないが起動方法は以下の通りである,

1. コマンドライン上にて `editor_learner` を入力する.
2. `editor_learner` を起動することでホームディレクトリに `editor_learner/workshop` と呼ばれるファイルが作成される. `workshop` は作業場という意味である.
3. `workshop` の中に `question.rb` と `answer.rb`, `random.h.rb` と `ruby_1` `ruby_6` が作成され, `ruby_1``ruby_6` の中に `1.rb` `3.rb` が作成されていることを確認する.

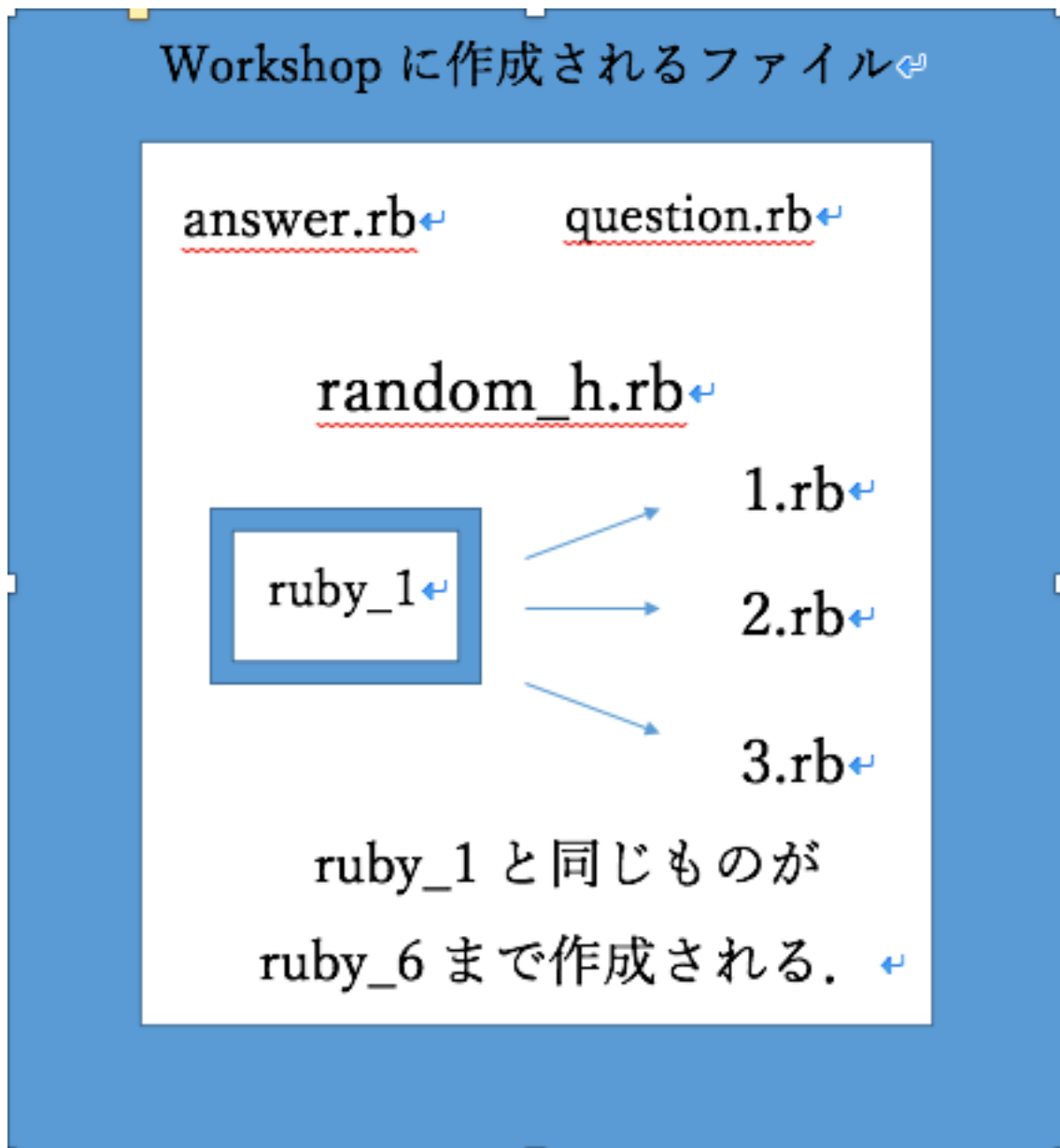


図 3.1: 作られるファイルの説明.

1. 起動すると以下のようなサブコマンドの書かれた画面が表示されることを確認する.

Commands:

```
editor_lerner delete [number~number]
```

```
editor_lerner help [COMMAND]
```

```
editor_lerner random_check
```

```
editor_lerner sequential_check [lesson_number] [1~3numbers]
```

2. `editor_learner` の後にサブコマンドと必要に応じた引数を入力すると動作する。それぞれのサブコマンドの更に詳しい説明は以下の通りである。

### 3.5 delete

`editor_learner` を起動することで初期設定で述べたようにホームディレクトリに `editor_learner/workshop` が作成される。delete は workshop に作成された `ruby_1`～`ruby_6` を削除するために作成されたものである。sequential\_check で1度プログラムを作成してしまふと再度実行すると `I have been finished!` と表示されてしまうので、削除するコマンドを作成しました。コマンド例は以下の通りである。

**`editor_learner delete 1 3`**

上記のように入力することで1～3までのファイルが削除される。サブコマンドの後の引数は2つの数字(char型)であり、削除するファイルの範囲を入力する。

### 3.6 random\_h.rb と sequential\_h.rb

`random_h.rb` と `sequential_h.rb` が初期設定で作成され、`editor_learner` を起動することで自動的に作成され、`random_check` と `sequential_check` を行う際に最初に関くファイルとなる。random\_check 用と sequential\_check 用に二つのファイルがある。random\_check 用のファイルは以下の通りである。



```
File Edit Options Buffers Tools Ruby Help
# to open question.rb
# c-x 2: split window vertically
# c-x c-f: find file and input question.rb
# open a.rb as above
# c-x 3: split window horizontally
# c-x c-f: find file and input answer.rb
# move the other window
# c-x o: other window
# then edit answer.rb as question.rb

# c-a: move ahead
# c-d: delete character
# c-x c-s: save file
# c-x c-c: quit edit
# c-k: kill the line
# c-y: paste of killed line
```

図 3.2: random\_h.rb.

上から順に説明すると、

1. question.rb を開くために c-x2 で画面を 2 分割にする。
2. c-x c-f で question.rb のパスを入力して開く。
3. 次に answer.rb を開くために画面を 3 分割する
4. 同様に c-x c-f で answer.rb のパスを入力して開く。
5. c-x o で answer.rb を編集するためにポインタを移動させる。
6. question.rb に書かれているコードを answer.rb に写す。

これらの手順が random\_h.rb に記述されている。全ての手順を終えたターミナルの状態は以下の通り、

```
File Edit Options Buffers Tools Ruby Help
# coding: utf-8
country = 'italy'

if country == 'japan'
  'こんにちは'
elsif country == 'us'
  'Hello'
elsif country == 'italy'
  'ciao'
else
  # to open question.rb
  # c-x 2: split window vertically
  # c-x c-f: find file and input question.rb
  # open a.rb as above
  # c-x 3: split window horizontally
  # c-x c-f: find file and input answer.rb
  # move the other window
  # c-x o: other window
  # then edit answer.rb as question.rb

numbers = [1, 2, 3, 4]
sum = 0
numbers.each do |n|
  sum += n
end
sum
```

図 3.3: 全ての操作を終えたターミナル画面.

上記の画像では、右上に問題である question.rb が表示され、それを左上にある answer.rb に写す形となる.

```
#to open q.rb
# c-x 2: find file and input q.rb
# c-x c-f: find file and input q.rb
# open 1-3.rb as above
# c-x 3: split find file and input 1-3.rb
# move the other window
# c-x o: other window
# then edit 1-3.rb q.rb

# c-a:move ahead
# c-d: delete character
# c-x c-s: save file
# c-x c-c: quit edit
# c-k: kill the line
# c-y: paste of killed line
```

図 3.4: sequential\_h.rb.

書かれている内容自体は random\_h.rb とほとんど差異がないが、開くファイルの名前が違うため別のファイルとして作成された. この手順に沿って作業することになる. 下に書かれているのは主要キーバインドであり、必要に応じて見て、使用する形となっている.

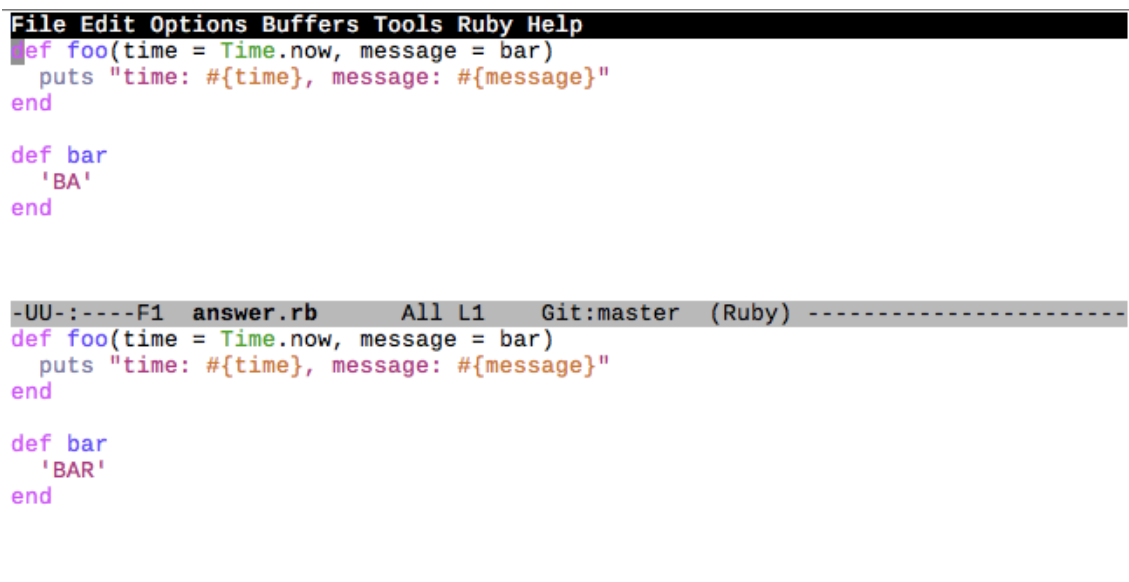
上記の手順を行なったターミナル画面の状態は random\_h.rb の最終形態を同じである。

### 3.7 random\_check の動作

random\_check の動作開始から終了は以下の通りである。

1. コマンドライン上にて editor\_learner random\_check を入力
2. 新しいターミナル (ホームディレクトリ /editor\_learner/workshop から始まる) が開かれる。
3. random\_h.rb を開いて random\_h.rb に沿って question.rb に書かれているコードを answer.rb に写す。
4. 前のターミナルに戻り、コマンドラインに”check” と入力することで正誤判定を行ってくれる。
5. 間違っていれば diff-lcs により間違った箇所が表示される。
6. 正しければ新しいターミナルが開かれてから終了までの時間と It have been finished! が表示され終了となる。

間違っていた場合の表示例は以下の通り、



```
File Edit Options Buffers Tools Ruby Help
def foo(time = Time.now, message = bar)
  puts "time: #{time}, message: #{message}"
end

def bar
  'BA'
end

-UU-:----F1  answer.rb      All L1   Git:master  (Ruby) -----
def foo(time = Time.now, message = bar)
  puts "time: #{time}, message: #{message}"
end

def bar
  'BAR'
end
```

図 3.5: 間違っている図,

```

"editor_learner-1.1.2, 1.1.1, 1.1.0, 1.0.3, 1.0.2, 1.0.1, 1.0.0, 0.1.2"
3
check starting ...
type following commands on the terminal
> emacs question.rb answer.rb
tab 1 of window id 10030
check
[["+", 133, "R"]]

```

図 3.6: 間違っている箇所の表示,

上記の図では question.rb の 'BAR' の部分が answer.rb では 'BA' になっている。よって 133 文字目に question.rb には R があるが、answer.rb には無いと表示されている。更に次回 random\_check 起動時には前に書いたコードが answer.rb に格納されたままなので全て削除するのではなく、前のコードの必要な部分は残すことができる。random\_check の大きな目的は typing 速度、正確性の向上、editor 操作や Ruby 言語の習熟に重点を置いている。いかに早く終わらせるかのポイントが typing 速度、正確性と editor 操作である。

## 3.8 sequential\_check の動作

sequential\_check の動作開始から終了は以下の通りである。

1. コマンドライン上で editor\_learner sequential\_check(1~6 の数字) (1~3 の数字) を入力
2. 新しいターミナル (ホームディレクトリ /editor\_learner/workshop/ruby\_(1~6 の数字)) が開かれる。
3. sequential\_h.rb を開いて sequential\_h.rb に沿って q.rb に書かれている内容を第 2 引数の数字.rb に写す。
4. 前のターミナルに戻り、コマンドラインに "check" と入力することで正誤判定を行う。
5. 間違っていれば間違った箇所が表示される。再度 q.rb と第 2 引数の数字.rb を開いて間違った箇所を修正する。
6. 正しければ ruby\_1/1.rb is done! のように表示される。

sequential\_check は 1~3 の順に 1.rb がリファクタリングや追加され 2.rb になり、完成形が 3.rb になるといった形式である。連続的なプログラムの完成までを写経するので sequential\_check と名付けられた。sequential\_check の大きな目的はリファクタリングによる Ruby

言語の学習と CUI 操作によるキーバインドの習熟, タイピング速度, 正確性の向上に重点を置いている. コードがリファクタリングされる様を写経することで自分自身で Ruby のコードを書くときに他の人が見やすくなるようなコードが書けるようになる.

## 第4章 実装コードの解説

本章では，今回作成したプログラムをライブラリ化し継続的な発展が可能なようにそれぞれの処理の解説を記述する．

### 4.1 起動時に動作するプログラム

`initialize` と名前に付けられたメソッドは特殊なメソッドでクラス内に記述した場合にはオブジェクトが作成されるときに自動的に呼び出される．`editor_learner` を動作したとき自動的に呼び出される部分である．

```

def initialize(*args)
  super
  @prac_dir="#{ENV['HOME']}/editor_learner/workshop"
  @lib_location = Open3.capture3("gem environment gemdir")
  @versions = Open3.capture3("gem list editor_learner")
  p @latest_version = @versions[0].chomp.gsub(' (', '-').gsub(')', '')
  @inject = File.join(@lib_location[0].chomp, "/gems/#{@latest_version}/lib")
  if File.exist?(@prac_dir) != true then
    FileUtils.mkdir_p(@prac_dir)
    FileUtils.touch("#{@prac_dir}/question.rb")
    FileUtils.touch("#{@prac_dir}/answer.rb")
    FileUtils.touch("#{@prac_dir}/random_h.rb")
    if File.exist?("#{@inject}/random_h.rb") == true then
      FileUtils.cp("#{@inject}/random_h.rb", "#{@prac_dir}/random_h.rb")
    elsif
      FileUtils.cp("#{ENV['HOME']}/editor_learner/lib/random_h.rb",
        "#{@prac_dir}/random_h.rb")
    end
  end
end

range = 1..6
range_ruby = 1..3
range.each do|num|
  if File.exist?("#{@prac_dir}/ruby_#{num}") != true then
    FileUtils.mkdir("#{@prac_dir}/ruby_#{num}")
    FileUtils.touch("#{@prac_dir}/ruby_#{num}/q.rb")
    FileUtils.touch("#{@prac_dir}/ruby_#{num}/sequential_h.rb")
    if File.exist?("#{@inject}/sequential_h.rb") == true then
      FileUtils.cp("#{@inject}/sequential_h.rb", "#{@prac_dir}/ruby_#{num}/sequential_h.rb")
    else
      FileUtils.cp("#{ENV['HOME']}/editor_learner/lib/sequential_h.rb",
        "#{@prac_dir}/ruby_#{num}/sequential_h.rb")
    end
  end
  range_ruby.each do|n|

```

```

        FileUtils.touch("#{@prac_dir}/ruby_#{num}/#{n}.rb")
      end
    end
  end
end

```

この部分は基本的にディレクトリやファイルの作成が主である。上から順に説明すると、`@prac_dir` はホームディレクトリ/`editor_learner/workshop` を指しており、ファイルを作る際のパスとして作成されたインスタンス定数である。その後の3つのインスタンス定数(`@lib_location`, `@versions`, `@latest_version`) は `gem` で `install` された場合ファイルの場所がホームディレクトリ/`.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0/gems` の `editor_learner` に格納されているため `gem` で `install` した人と `github` で `install` した人とはパスが変わってしまうためこれらの3つのインスタンス定数を利用し `@inject` を用意した。これにより、`gem` から `install` した場合でも正しく動作するようになった。実際の振る舞いとしては、`File.exist` により `prac_dir` がなければディレクトリを作成しさらにその中に `question.rb` と `answer.rb` を作成する。`gem` にリリースしていることから `gem` で `install` した人と `github` で `install` した人のパスの違いを `if` 文で条件分岐させている。これにより `random_h.rb` や `sequential_h.rb` を正常にコピーすることができた。

#### 4.1.1 プログラム内のインスタンス変数の概要

‘@’で始まる変数はインスタンス変数であり、特定のオブジェクトに所属している。インスタンス変数はそのクラスまたはサブクラスのメソッドから参照できる。初期化されない孫スタンス変数を参照した時の値は `nil` である。

このメソッドで使用されているインスタンス変数は5つである。`prac_dir` はホームディレクトリ/`editor_learner/workshop` を指しており、必要なファイルをここに作るのもっとも基本的なパスとして受け渡すインスタンス変数となっている。その後の4つのインスタンス変数は `gem` から `install` した場合における、`editor_learner` が格納されているパスを受け渡すためのインスタンス変数である。一つずつの説明は以下の通り、

**lib\_location** ターミナル上で”`gem environment gemdir`”を入力した場合に出力されるパス



を格納している。(自分のターミナル場で実行すると/Users/souki/.rbenv/versions/2.4.0/lib/ruby/gems/2.4.0) 最初は system"gem environment gemdir"に代入していたが system の返す値は文字列ではなく'true'と'false'である。なので、ここで open3 が必要となった。

**versions** gem で install された editor\_learner の version を受け取るためのパスを格納したインスタンス変数である。Ruby の version が 2.4.0 以上であるならば、正しく最新版のパスを入手できるが、それ以下だと正しくパスを入手できないので Ruby の version が 2.4.0 以上必要となる。version をあげたくない人には editor\_learner の gem を最新版のみ install していれば正しくパスを受け取ることができる。

**latest\_versions** versions で受け取った editor\_learner の version の最新部分のパスを格納したインスタンス変数である。

**inject** 実際にこれらのパスをつなぎ合わせてできる gem で install された editor\_learner が格納されているパスが格納されているインスタンス変数である。(自分の場合は /Users/souki/.rbenv /versions/2.4.0/lib/ruby/gems/2.4.0/gems/editor\_learner-1.1.2 となる)

inject はいろんなメソッドで使われるため、普通の変数では他のメソッドで呼び出せないためインスタンス変数で書かれている。

#### 4.1.2 File の作成

全てのパスの準備が整ったら実際に作業する場所に必要なファイル (question.rb や answer.rb など) の作成が行われる。本研究の initialize メソッドでは editor\_learner/workshop がホームディレクトリになれば作成する。さらに、その中に random\_check に必要なファイル (question.rb, answer.rb, random\_h.rb) が作成される。random\_h.rb は gem で install した場合は editor\_learner の格納されている部分からコピーを行なっている。次に、sequential\_check に必要なファイルを作成する。editor\_learner/workshop に ruby\_1~ruby\_6 がなければ作成し、その中に 1.rb~3.rb と q.rb(問題をコピーするためのファイル) と sequential\_h.rb が作成される。sequential\_h.rb は random\_h.rb と同じで gem から install した場合は editor\_learner の格納されている部分からコピーを行なっている。このメソッドの大き

な役割はファイル作成であり、この部分がなければ全てのプログラムが動作しない。

## 4.2 ファイル削除処理 delete

sequential\_check で終了した chapter とその中の問題 (1.rb~3.rb) をもう一度行いたい場合に一度書き込んであるので「(例)ruby\_1/1.rb is done!」と表示されてしまう。なので終わった問題を再度やりたい場合には一度ファイルを削除しなければいけないので、delete メソッドを作成した。delete メソッドの大きな役割は sequential\_check で打ち込みが完了したファイルの削除である。ソースコードは以下の通り、

```
desc 'delete [number~number]', 'delete the ruby_file choose number to delete file'

def delete(n, m)
  range = n..m
  range.each{|num|
    if File.exist?("#{@prac_dir}/ruby_#{num}") == true then
      system "rm -rf #{@prac_dir}/ruby_#{num}"
    end
  }
end
```

コード自体はいたってシンプルで引数を2つ受け取ることでその間の範囲の File を削除するようなコードとなっている。system の”rm -rf ファイル名”がファイルを削除するコマンドとなっている。まず、そのファイルが存在しているかどうかの判定を File.exist で行う。そして、存在しているなら引数として受け取った n と m が範囲となっておりその範囲で for 文の働きをする each メソッドを使用することで削除を再帰的行なっている。

## 4.3 random\_check

random\_check のコードは以下の通り、

```

desc 'random_check', 'random check your typing and edit skill.'
def random_check(*argv)
  random = rand(1..15)
  p random
  s = "#{random}.rb"
  puts "check starting ..."
  puts "type following commands on the terminal"
  puts "> emacs question.rb answer.rb"

  src_dir = File.expand_path('../..', __FILE__)
  # "Users/souki/editor_learner"
  if File.exist?("#{@inject}/random_check_question/#{s}") == true then
    FileUtils.cp("#{@inject}/random_check_question/#{s}",
      "#{@prac_dir}/question.rb")
  elsif
    FileUtils.cp(File.join(src_dir, "lib/random_check_question/#{s}"),
      "#{@prac_dir}/question.rb")
  end
  open_terminal

  start_time = Time.now
  loop do
    a = STDIN.gets.chomp
    if a == "check" && FileUtils.compare_file("#{@prac_dir}/question.rb",
      "#{@prac_dir}/answer.rb") == true then
      puts "It have been finished!"
      break
    elsif FileUtils.compare_file("#{@prac_dir}/question.rb",
      "#{@prac_dir}/answer.rb") != true then
      @inputdata = File.open("#{@prac_dir}/answer.rb").readlines
      @checkdata = File.open("#{@prac_dir}/question.rb").readlines
      diffs = Diff::LCS.diff("#{@inputdata}", "#{@checkdata}")
      diffs.each do |diff|
        p diff
      end
    end
  end
end
end

```

```
end_time = Time.now  
time = end_time - start_time - 1  
  
puts "#{time} sec"  
end
```

random\_check の概要を簡単に説明すると 15 個ある Ruby のコードから 15 の乱数を取得し、選ばれた数字のファイルが問題としてコピーされて、それを answer.rb に入力することで正解していたら新しいターミナルが開かれてから終了までの時間を評価する仕組みとなっている。

上から解説を行うと、1~15 の random な乱数を取得、起動と同時に選ばれた数字のファイルを表示する。そして、src\_dir でホームディレクトリ/editor\_learner のパスが代入される。File.expand\_path() は、

第 2 引数のパスを起点に、第 1 引数のパスを、フルパスに展開するメソッドである。\_\_FILE\_\_ にはそのファイルのパスが入っているが、これは絶対パスとは限らない、そこで、File.expand\_path() の第 2 引数にこれを指定し、第 1 引数にそのファイルからの相対パスを書くことで、目的のファイルの絶対パスが得られる [8]。

ここで相対パスを得ることで File.join で必要なパスと結合させる。そして、gem で install した人と github から clone した場合によるファイルのパスの違いを if で条件分岐。そして、15 個あるソースコードから乱数によって選ばれたファイルが question.rb にコピーされる。コピーされた後に新しいターミナルが開かれ、時間計測が開始する。そして、answer.rb に question.rb と同じ内容を書き込む。FileUtils.compare\_file を使うことで tab などによる空白などもきちんと判定する。そうしなければ、改行などをめちゃくちゃにしても正しいと判断されてしまうからである。見にくいプログラムを書くと、他の人が見たとき理解しづらいので改行などもきちんと判定している。全てうち終えることで、前のターミナルに戻り”check”と入力する。check を前のターミナルに入力できるように gets を使った。初めに gets だけを使用した時改行が入ってしまいうまく入力できなかった。しかし、chomp を入れることで改行をなくすことに成功。しかし、argv と gets を併用することが不可能なことが判明した。そこで gets の前に STDIN を入れることで argv との併用が可能ながわ

り, `STDIN.gets.chomp` と入力することで `argv` との併用が可能となり, キーボードからの入力を受け取ることができた. そして, `check` が入力されてかつ `FileUtils.compare_file` で比較の結果が正しければ時間計測を終了し”it have been finished!”を表示する. 間違っていた場合は `file.open` により `question.rb` と `answer.rb` を `input` と `output` に格納されて `Diff::LCS` の `diff` によって間違っている箇所だけを表示する. 間違った部分を見て, 再度コードを打ち直し正しいと判定されると `break` で終了する. 一連の流れは異常である.

## 4.4 sequential\_check

`sequential_check` の場合はリファクタリングにあたりたくさんのインスタンス定数を作った. コードは以下の通り,

```

desc 'sequential_check [lesson_number] [1~3number] ',
'sequential check your typing skill and edit skill choose number'
def sequential_check(*argv, n, m)
  l = m.to_i - 1

  @seq_dir = "lib/sequential_check_question"
  q_rb = "ruby_#{n}/#{m}.rb"
  @seqnm_dir = File.join(@seq_dir, q_rb)
  @pracnm_dir = "#{ENV['HOME']}/editor_learner/workshop/ruby_#{n}/#{m}.rb"
  @seqnq_dir = "lib/sequential_check_question/ruby_#{n}/q.rb"
  @pracnq_dir = "#{ENV['HOME']}/editor_learner/workshop/ruby_#{n}/q.rb"
  @seqnl_dir = "lib/sequential_check_question/ruby_#{n}/#{l}.rb"
  @pracnl_dir = "#{ENV['HOME']}/editor_learner/workshop/ruby_#{n}/#{l}.rb"
  puts "check starting ..."
  puts "type following commands on the terminal"
  src_dir = File.expand_path(' ../../..', __FILE__)
  if File.exist?("#{@inject}/sequential_check_question/ruby_#{n}/#{m}.rb")
    == true then
      FileUtils.cp("#{@inject}/sequential_check_question/ruby_#{n}/#{m}.rb"
        , "#{@pracnq_dir}")
    elsif
      FileUtils.cp(File.join(src_dir, "#{@seqnm_dir}"), "#{@pracnq_dir}")
    end
  if l != 0 && FileUtils.compare_file("#{@pracnm_dir}", "#{@pracnq_dir}")
    != true
    FileUtils.compare_file("#{@pracnl_dir}", (File.join(src_dir,
      "#{@seqnl_dir}"))) == true
    FileUtils.cp("#{@pracnl_dir}", "#{@pracnm_dir}")
  end

  if FileUtils.compare_file(@pracnm_dir, @pracnq_dir) != true then
    system "osascript -e 'tell application \"Terminal\" to do script
      \"cd #{@prac_dir}/ruby_#{n} \" ' "
    loop do
      a = STDIN.gets.chomp
      if a == "check" && FileUtils.compare_file("#{@pracnm_dir}",
        "#{@pracnq_dir}") == true then
        puts "ruby_#{n}/#{m}.rb is done!"
        break
      end
    end
  end
end

```

```

    elsif FileUtils.compare_file("#{@pracnm_dir}", "#{@pracnq_dir}")
      != true then
        @inputdata = File.open("#{@pracnm_dir}").readlines
        @checkdata = File.open("#{@pracnq_dir}").readlines
        diffs = Diff::LCS.diff("#{@inputdata}", "#{@checkdata}")
        diffs.each do |diff|
          p diff
        end
      end
    end
  end
else
  p "ruby_#{n}/#{m}.rb is finished!"
end
end
end

```

#### 4.4.1 インスタンス定数に格納されたパス

インスタンス定数に格納されているパスについての説明は上から順に以下の通り、

1. seq\_dir は github で clone した人が問題をコピーするときに使うパスである。
2. seqnm\_dir はその名の通り seq\_dir に引数である n と m を代入したパスである。例として引数に 1 と 1 が代入された時は以下の通り、
  1. editor\_learner/sequential\_check\_question/ruby\_1/1.rb となる。
3. pracnm\_dir は prac\_dir に二つの引数 n と m を代入したものである。実際に作業するところのパスとして使用する。例として引数として 1 と 1 が代入された時は以下の通り、
  1. ホームディレクトリ/editor\_learner/workshop/ruby\_1/1.rb が格納される。
4. 同様に seq と prac の後についている文字はその後の ruby\_(数字)/(数字).rb の数字に入る文字を後につけている。
5. 例として、seqn\_dir は ruby\_n/1.rb のことである。以下、同じように引数の2つの数字が seq と prac の後についている。

### 4.4.2 動作部分

まず gem で install した場合と github で install した場合による違いを条件分岐によりパスを変えている。さらに 1.rb が終了していた場合 2.rb に 1.rb をコピーした状態から始まるように処理が行われている。理由は前のコードがどのように変わったか、リファクタリングされたかがわかりやすいように前のコードをコピーしている。その後は”check”が入力された時かつ FileUtils.compare\_file で正しければ終了。間違っていれば Diff::LCS で間違っている箇所を表示。もう一度修正し、”check”を入力、正解していれば終了。 ”check”による正誤判定などは random\_check と同様の書き方をしている。以上が一連のコードの解説である。

## 4.5 新しいターミナルを開く open\_terminal

新しいターミナルを開くメソッドである。コードは以下の通りである。

```
def open_terminal
  pwd = Dir.pwd
  system "osascript -e 'tell application \"Terminal\" to do script \"\n\ncd #{@prac_dir} \" \"'\n"
end
```

新しく開かれたターミナルは prac\_dir(editor\_learner/workshop) のディレクトリからスタートするように設定されている。 random\_check では editor\_learner/workshop でターミナルが開かれ、 sequential\_check では editor\_learner/workshop/第1引数で入力されたファイルの場所が開かれるようになっている。



## 第5章 他のソフトとの比較

他のタイピングソフトとの比較を行った表が以下の通りである。

表 5.1: 他のソフトとの比較.

	UI	プログラムの実行	タイピング以外の付加価値	タイピング文字
editor_learner	CUI	可能	editor操作	プログラミング言語
PTYPING	GUI	不可能	プログラミング言語が豊富	プログラミング言語
e-typing	GUI	不可能	資格取得の練習	ローマ字
寿司打	GUI	不可能	ランキング登録可能	ローマ字

上記のタイピングソフトは自分もよく使っていたタイピングソフトであり、評価も高いソフトである。それぞれの特徴は以下の通り。

1. PTYPING: 豊富なプログラミング言語がタイピング可能
2. e-typing: 資格取得にもつながる練習が可能。間違いが多い箇所を指摘してくれる。
3. 寿司打: 自分が一番よく使ったソフト、GUI ベースで飽きずに継続しやすい。

それぞれの特徴があるが、人気ソフトの中でもプログラムの実行が可能なソフトは発見できなかった。プログラマにとってコードを書いて実行しないのは、テストを受けて結果を見ないのと同義である。また、これらのソフトは全て Web 上で行なっており、editor は全く使わない。プログラマにとってコードだけ書いて editor を使って実行しないことなどない。よっていかに editor\_learner がプログラマ向けのソフトかが容易にわかる。

## 5.1 考察

これら全てのソフトを利用した結果、editor\_learner はローマ字入力ができない点では他のソフトに遅れをとるが、実際にプログラムを書くようになってからコードを写経することで {} や () などといった約物の入力が非常に早くなった。さらに、editor\_learner は現段階では Ruby の学習のみだが、引数を変えて元となるプログラムを作成することで全てのプログラム言語を学ぶことができる。さらに、実際にコードを入力することができるソフトはたくさんあるが、実行可能なものは少ない (Web で行うものが大半を占めているから)。実際に西谷研究室で editor\_learner で学習を行っていない学生と行った自分の random\_check 平均秒数は前者は 200 秒程なのに対して、自分は 60 秒程である。これらの結果から editor\_learner による学習により、Ruby 言語の学習にもなり、タイピング速度、正確性の向上、CUI 操作の適応による差が出たと考えた。

## 第6章 総括

実際に今までたくさんのタイピングソフトやプログラムコードの打てるタイピングソフトを数多く利用してきたが，editor 操作の習熟が可能なソフトは見たことも聞いたこともなかった．実際にタイピングだけが早い学生はたくさんいるがeditor 操作やキーバインドも使いこなせる学生は少なかった．本研究で開発した editor\_learner によりそれらの技術も上達し，作業効率などの向上が期待される．

## 第7章 謝辞

本研究を行うにあたり，終始多大なるご指導，御鞭撻をいただいた西谷滋人教授に対し，深く御礼申し上げます。また，本研究の進行に伴い，様々な助力，知識の供給をいただきました西谷研究室の同輩，先輩方に心から感謝の意を示します。本当にありがとうございました。

# 参考文献

- [1] Andrew Hunt, David Thomas, 「達人プログラマー」, (オーム社, 2016 年).
- [2] Ruby ホームページ, <https://www.ruby-lang.org/ja/>, accessed 2018.2.8.
- [3] S. Koichiro, Rubygems のススメ, <https://qiita.com/sumyapp/items/5ec58bf3567e557c24d7>, accessed 2018.2.8.
- [4] Weblio, キーバインド, <https://www.webl.io.jp/content/キーバインド>, accessed 2018.2.8.
- [5] Weblio, CUI, <https://www.webl.io.jp/content/CUI> , accessed 2018.2.8.
- [6] S. Kouichiro, Thor の使い方, <https://qiita.com/succi0303/items/32560103190436c9435b>, accessed 2018.2.8.
- [7] SideCI, Rubocop の使用, <http://blog-ja.sideci.com/entry/2015/03/12/160441>, accessed 2018.2.9
- [8] Ruby, 相対パス, <https://www.xmisao.com/2013/11/21/ruby-require-relative.html>, accessed 2018.2.8.