# コンパイラ再評価者向け演習: cit-compiler-special

立合もしくは監督の方は、演習実施前に再評価演習の受講者へ心得の指示をお願いします。 受講生は同指示に従うこと。

## 1. 演習の概要

これは、コンパイラの再評価者向けの演習のAssignmentです。

1. Lox言語でプログラムを作成して動かす
2. 自分のLoxインタプリタを動作させる

## 2. 演習の準備

基本的には講義中の演習を実施していたのであれば、これらの準部はすでに済んでいるはず。 もし、講義中に準備していなかったものがあれば、最初にこれらを準備すること（していなければ演習そのものが実施できない）。

### 2.1. 参照する資料

- Googleクラスルームの「コンパイラ」の講義資料

演習の進め方や、Asignmentの受け取り方、必要なソフトウェア等については、Boxフォルダで配布した次の文書を参照のこと。

**コンパイラの講義用Boxフォルダの共有リンク**
https://nihon-u.box.com/s/5805inwn3t7442fq1wix2qyay3cfrtiq

- asciidocプラグインの設定.pdf（Chromeブラウザ用、Visual Studio Code用）
- GitHub_Classroom_Assignmentの受け取り方_20220920.pdf
- gitコマンドの使い方.pdf
- github-git-cheat-sheet.pdf

念のため、Boxへ移行する以前のGoogleドライブも挙げておく。

**コンパイラの講義用Googleドライブの共有リンク**
https://bit.ly/4fTCkI8

もし、Googleドライブが閲覧できないようなら、講義中に指示した共有の申請をしていないのが原因。 共有申請をすればよいが、電子メールベースなので、応答には時間がかかる。

### 2.2. このAssignmentを受理（Accept）する

先述の「GitHub_Classroom_Assignmentの受け取り方_20220920.pdf」を参照して、このAssignmentを受理しなさい。

受理すると、自分のGitHubアカウントのホームディレクトリに「cit-compiler-special-アカウント名」のようなリポジトリが作られる。

### 2.3. ソフトウェアのインストール

下記のソフトウェアをインストールする。

#### 2.3.1. Visual Studio Code

- https://azure.microsoft.com/ja-jp/products/visual-studio-code
- 機能拡張から、Java用、lox用の機能拡張を入れておくとよい。

#### 2.3.2. Git

（Windowsを想定し）Git For Windowsをインストールする。

インストーラ実行中の設定で `Git Bash` の追加を忘れないこと。

- 次のページを参照して、インストールする。
  - https://gitforwindows.org/
- わからないときは、次のページを参考にする。
  - https://www.curict.com/item/60/60bfe0e.html

Gitがインストールできたか確認する。

1. コマンドプロンプト（Git Bashでもよい）を開く。
2. `git --version` を実行する。
   - `git verion 2.46.0.windows.1` などバージョン番号が表示されればよい。

### 2.3.3. Git Bashの確認

GitBashの設定を調整する。

1. Windowsのタスクバーのスタートメニューの検索欄に「Git Bash」を入力して検索する。
2. ポップアップしたスタートアップメニューのアプリ欄に表示される「Git Bash」をクリックして起動する。
3. Git Bashのターミナルが表示される。
4. ターミナルの左上のアイコンをクリックし、ポップアップメニューを表示する。
5. ポップアップしたメニューから「Options」をクリックして、Optionsダイアログを表示する。
6. 左の「Looks」欄を選択し、右に表示された項目の「Theme」で「luminous」など、背景が明るいテーマから好みのものを選択する。
7. 左の「Text」欄を選択し、右に表示された項目の「Font」で「Select」ボタンをクリックし、日本語のフォントを選択、フォントサイズを少し大きめに変更する。
8. 右下の「Save」ボタンをクリックして設定を保存する。

下記実行例の `$` はターミナルが表示するプロンプトを表す。 **自分の入力ではないので入力しないこと。**

ターミナルのプロンプトは、実行する環境によって少しずつ異なる。 たとえば、ある環境では、実際に表示されてるプロンプトは、

```
kuboaki@LAPTOP-URPL5RT3 MINGW64 /y/nichidai/cit-complier-special (main)
$ （ここで入力待ち）
```

のようになっていた。

確認の実行例（1）

```
$ cd /y/Users/kuboaki/cit-compiler-special  1
$ pwd  2
/y/Users/kuboaki/cit-compiler-special  3
$ ls  4
README.adoc codes/ css/ image_size_matter.adoc images/  5
```

1 `cd` はディレクトリの移動コマンド。この例では、 `y` がドライブ名、後に続くのが演習のAssignmentのあるディレクトリ。

2 `pwd` は、現在のディレクトリを表示するコマンド。

3 ディレクトリが移動できているか確認した。

4 `ls` コマンドで現在のディレクトリのファイルとディレクトリをリストする。

5 `README.adoc` やその他のファイルやディレクトが含まれていることを確認した。

確認の実行例（2）

```
$ head -50 README.adoc  1

:linkcss:
:stylesdir: css
:stylesheet: mystyle.css
（中略）
== 演習の準備  2

[IMPORTANT]
--
基本的には講義中の演習を実施していたのであれば、これらの準部はすでに済んでいるはず。
もし、講義中に準備していなかったものがあれば、最初にこれらを準備すること（していなければ演習そのものが実施できない）。
--

=== 参照する資料
```

1 `head` は、テキストファイルの先頭を表示するコマンド。 `-50` オプションで、先頭から50行目までをから表示するよう指示している。

2 Git Bashのターミナルで日本語が文字化けしていないことを確認した。

### 2.3.4. Java開発環境（JDK）

Oracle JDKまたはOpenJDKをインストールする

- 次のページを参照して、インストールする。
  - https://www.oracle.com/java/technologies/downloads/
  - http://jdk.java.net/
- VS Codeを使う人は下記を参照してプラグインを導入するとよい
  - https://note.com/liber_grp/n/n88f3f0a6fdf1
- Eclipseを使う人は下記を参照してPleiades All in One Eclipseを導入するとよい
  - https://willbrains.jp/index.html#/pleiades_distros2024.html

インストールできたか確認する。

古いJDKが入っていると、演習のプログラムが「コンパイルできない・動作しないとい」ったことが起きる。 古いものがあれば削除し、それから新しいものをインストールすること。

確認の実行例

```
$ java --version  1
openjdk 21.0,6 2025-01-21 LTS  2
（略）
#javac --version  3
javac 21.0.6  4
```

1 `java` コマンドは、Javaのコンパイル済みプログラム（ `.class` や `.jar` ファイル）を指定して実行するインタプリタ。 `--version` オプションでバージョンを表示している。

2 自分がインストールしたJDKのバージョンと一致するか確認する。

3 `javac` コマンドは、Javaのソースコード( `.java` ファイル）のコンパイラ。 `--version` オプションでバージョンを表示している。

4 自分がインストールしたJDKのバージョンと一致するか確認する。

## 2.4. GitHubアカウントの作成

GitHubクラスルームを使うためには、日大のメールアカウントで作成したGitHubアカウントが必要。

もし、日大のメールアカウントを使用したGitHubアカウントを作成していないなら、下記ページを参照して、個人用アカウント作成すること。

**GitHub でのアカウントの作成**
https://docs.github.com/ja/get-started/start-your-journey/creating-an-account-on-github

## 2.5. GitHubの Personal Access Token の作成

GitHubはベーシック認証（ユーザー名とパスワード）をやめている。 その代わりとして、コマンドラインからGitHubにアクセスするときにパスワード入力を求められたら、パスワードの代わりにあらかじめ作成しておいた Personal Access Token を使う。

下記手順を参考にして、 Personal Access Token を作成する。

1. PCでウェブブラウザを使ってGitHubにログインする。
2. ページ右上の自分のアカウントを示すアイコンをクリックしてポップアップメニューを表示する。
3. ポップアップしたメニューを下記のようにたどり、 `Personal access tokens (classic)` ページへ移動する。
   - `Settings ＞Developer settings＞Personal access tokens＞Tokens(classic)`
4. `Generate new Token` ボタンをクリックして、新しいトークンを作成する。
5. トークン名をつけるよう求められたら、 `for_pc_cli` など自分で用途がわかる名前をつけておく。

作成したアクセストークンは、必ずすぐに分かる場所（デスクトップ等）に電子的に保管しておくこと。
コマンドラインからGitHubを使うときに、パスワードの代わりに入力できるよう備えておく。

## 2.6. プレビューアの準備と確認

このファイル `README.adoc` のような Asciidoctor 形式のファイルをプレビューするため、プレビューアを用意する。

1. Chromeブラウザをインストールしていないなら、インストールする。
2. Chromeブラウザのウィンドウの右上、縦向きの ⋯ で示されるハンバーガーメニューを表示する。
3. ポップアップしたメニューを下記のようにたどり、「Chromeウェブストアにアクセス」ページへ移動する。
   - 機能拡張＞Chromeウェブストアにアクセス
4. 「機能拡張とテーマを検索」欄に「Asciidoctor」と入力して検索する。
5. 「Asciidoctor.js Live Preview」という機能拡張が見つかったら、「Chromeに追加」する。
   - Chromeブラウザのメニューアイコンのリストに Asciidoctorのアイコン（赤地の四角に`A`のアイコン）が追加される。
6. 追加した機能拡張のアイコンを右クリックしてポップアップメニューを表示する。
7. ポップアップしたメニューから「オプション」をクリックして、オプション設定のページへ移動する。
8. ポップアップしているダイアログの「Asciidoctor options」の「Custom attributes」設定に下記を追加する。
   - `:sourcesdir: codes`
9. 追加できたら、ダイアログウインドウを閉じ、裏で表示されている機能拡張の設定ページに戻る。
10. 設定項目の中から「ファイルのURLへのアクセスを許可する」を探し、設定をONにする。
    - それでもchromeでローカルファイル（file:///で始まるファイル）にアクセスできない場合は、Chromeウェブストアから「GoogleChrome でローカルファイルを開く」を追加する。
11. このファイル `README.adoc` をChromeブラウザで開いて、プレビューできることを確認する。
    - 追加した機能拡張のアイコンを左クリックするたびに、プレビュー表示と元のadocファイルの表示が切り替わる。

もし、Chromeブラウザでadocファイルのプレビューが表示できない場合は、プレビューにこだわらずに、テキストエディタでテキストファイルとして参照する。

## 2.7. 作業ディレクトリを作成する

ここでは、 Git Bash ターミナルを使って作業することに注意。

演習用作業ディレクトリを作る

```
$ cd  1
$ mkdir compiler-work  2
$ cd compiler-work  3
```

1 自分のホームディレクトリに移動する（引数なしの `cd` はホームへの移動）。

2 `compiler-work` という名前のディレクトリを作る。

3 作成したディレクトリへ移動する。

既存の他のリポジトリのディレクトリの中に作業用ディレクトリを作らないこと。 リポジトリの入れ子があると、commitやpushでpermission エラーになることがある。

```
✅
├ lox01-AAA-BBB
├ cit-compiler-work  1

❌
├ lox01-AAA-BBB
        ├ cit-compiler-work  2
```

1 **OK**: 既存のリポジトリとは独立した場所に作成した。

2 **NG**: 既存のリポジトリである `lox01-AAA-BBB` の中に別のリポジトリを作ってしまった。

## 2.8. このAssignmentのリポジトリをCloneする

作成した作業ディレクトリにこのAssignmentのリポジトリをCloneしなさい。

このAsaignmentをCloneするURLを取得する

1. ウェブブラウザで、このAsaignmentのGitHubリポジトリページを開く。
2. ページ上部にある「<> Code」という緑のボタンをクリックして、ポップアップメニューを開く。
3. ポップアップしたメニューから「Local」タブ、「HTTPS」タブを選択し、「Clone using the web URL」の上の欄のURLをコピーする。

演習用作業ディレクトリにAsaignmentをCloneする

```
$ pwd  1
（どこかの）/compiler-work
$ git clone （取得したURL）  2
$ cd （リポジトリ名）  3
```

1. 作業用ディレクトリ `compiler-work` にいることを確認した。
2. `git clone` コマンドを使って、自分のAssignmentのリポジトリをCloneする。
3. cloneしたディレクトリへ移動する。

# 3. 課題1

作成済みのLoxインタプリタ（jLox）を使って、Loxのプログラムを動かしなさい。

cit-compiler-specialのディレクトリ構成

```
cit-compiler-special
├──── README.adoc
├──── codes/
├──── css/
├──── image size_matter.adoc
├──── images/
└──── jlox/
```

## 3.1. 作成済みのjLoxを動かす

1. このAssignmentの `jlox` ディレクトリへ移動する。
2. jloxプログラムをコンパイルする。
3. ソースディレクトリにclassファイルができているか確認する。
4. jloxプログラムを起動する。
5. サンプルプログラムを動かす。

実行例

```
$ cd jlox  1
$ ls
com/  2
$ javac com/craftinginterpreters/lox/Lox.java  3
$ ls com/
$ ls com/craftinginterpreters/lox/*.class  4
 com/craftinginterpreters/lox/Environment.class
  （略）
 com/craftinginterpreters/lox/TokenType.class
$ java com.craftinginterpreters.lox.Lox  5
Welcom! Lox interpreter(jLox)
> print 1 +2;  6
3
> ^C  7
$
```

1. `cd` コマンドで、このAssignmentの `jlox` ディレクトリへ移動した。
2. `ls` コマンドで、ディレクトリのリストを表示した。
3. `javac` コマンドを使って、ソースコードをコンパイルした。
4. `ls` コマンドで、classファイルができているか確認した。
5. `java` コマンドを使って、 jLox を起動した。
6. 実行したいプログラムを入力し、結果が表示されるのを確認した。
7. `Ctrl + C` （Ctrlキーと`C`キーを同時に押して）インタプリタを停止した。

## 3.2. 任意の場所でjLoxを動かす

他のディレクトリで jLox を利用したいときは、コマンドラインの `-cp` オプションを使って classpath を指定する。

たとえば、 このAssignmentの `codes` ディレクトリへ移動してから実行したいとするときは、次のようにする。

別のディレクトリでloxインタプリタを実行する

```
$ pwd
（どこかの）/cit-compiler-special/jlox
$ cd ../codes  1
$ ls  2
breakfast02.bnf          sample01.lox  tiny_lang.bnf
personal_infomation.csv  sample02.lox
```

```
$ java -cp ../jlox com.craftinginterpreters.lox.Lox sample02.lox  3
60
false
CCCccc
true
0.5
```

1. cd コマンドを使って、 codes ディレクトリへ移動した。
2. このディレクトリに sample02.lox があることを確認した。
3. java コマンドに -cp オプションで jlox の場所を指定し、 sample02.lox を実行した。

## 3.3. Loxプログラムを作成して動かす

次のLoxプログラム breakfast.lox を codes ディレクトリに作成しなさい。

作成するLoxプログラム（breakfast.lox）

```
class Breakfast {
  init(meat, bread) {
    this.meat = meat;
    this.bread = bread;
  }
  cook() {
    return "Cooking now ... " + this.meat + " and " + this.bread + ".";
  }
  serve(who) {
    return "Enjoy your breakfast(" + this.meat + ", " + this.bread + "), " + who + ".";
  }
}

var breakfast = Breakfast("sausage", "toast");
print breakfast.cook();
print breakfast.serve("Gentleman");
print clock();
```

作成できたら、 3.2 に示した方法で実行しなさい。

breakfast.lox を実行する

```
$ pwd  1
（どこかの）/cit-compiler-special/coedes
$ ls  2
breakfast.lox breakfast02.bnf  personal infomation.csv
sample01.lox  sample02.lox  tiny_lang.bnf
```

```
$ java -cp ../jlox com.craftinginterpreters.lox.Lox breakfast.lox  3
（実行結果）
```

1. pwd コマンドを使って、 codes ディレクトリにいることを確認した。
2. このディレクトリに breakfast.lox があることを確認した。
3. java コマンドに -cp オプションで jlox の場所を指定し、 breakfast..lox を実行した。

次のページを参考に、breakfast.lox をターミナル上で実行している様子のスクリーンショットを撮りなさい。

**Windowsのスクリーンショットを撮る4つの方法**
https://www.pc-koubou.jp/magazine/35994?srsltid=AfmBOorbVX7AYZLr4Vg7F5Zrp-4U3SbxxGypobX_REF7D-jCkLfKsEw9

取得したスクリーンショットを images ディレクトリの breakfast.png と置き換えなさい。 その後、このページをリロードすると、下図が取得したスクリーンショットと置き換わったことを確認しなさい。

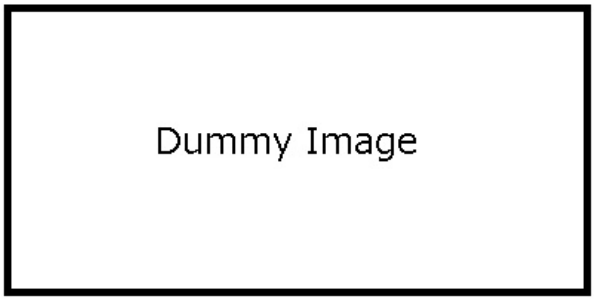


図 1. breakfast.loxを実行した結果

## 3.4. 作業を保存する

このAsainnmenntのリポジトリをコミットして、プッシュしなさい。

```
$ pwd  1
（どこかの）/cit-compiler-special/
$ git add .  2
$ git status  1
```

```
$ git commit -m "update"  3
$ git status  4
$ git push  5
```

1. `git status` コマンドを使って、現在のディレクトリを確認。もし、リポジトリの外のディレクトリにいたなら、ここに移動してくる。
2. `git add` コマンドを使って、更新対象のファイルをすべてステージに追加する（コミットの対象として指定する）。
3. `git commit` コマンドを使って、更新状態を保存する。 `-m` の後ろはコミットメッセージ。
4. コミットできたか確認した。
5. `git push` コマンドで、GitHub上のリポジトリへ更新分を反映する。

# 4. 課題2

Javaを使って、講義で実施した範囲をカバーするLoxのインタプリタを作りなさい。

## 4.1. 開発用ディレクトリを作成する

開発用ディレクトリを作成する

```
$ pwd  1
（このAssignmentのディレクトリ）
$ cd codes  2
$ mkdir com  3
$ mkdir com/craftinginterpreters
$ mkdir com/craftinginterpreters/lox
$ touch com/craftinginterpreters/lox/Lox.java  4
```

1. 現在のディレクトリがAssignmentのディレクトリであることを確認した。
2. このAssignmentのリポジトリの `codes` ディレクトリへ移動する。
3. `mkdir` コマンドを使って、 `codes` ディレクトリに下記のディレクトリ階層を作成する。
4. `touch` コマンドを使って、 `com/craftinginterpreters/lox/` ディレクトリに `Lox.java` ファイルを作った。

## 4.2. テキストの4章を作成する

1. Visual Studio Codeなどのテキストエディタで、4章までのコードを打ち込み、保存しなさい。
2. 次の実行例に従ってプログラムをコンパイルし、実行しなさい。

実行例

```
(codesディレクトリにいる)
$ javac com/craftinginterpreters/lox/Lox.java  1
$ ls com/craftinginterpreters/lox/*.class  2
 com/craftinginterpreters/lox/Environment.class
   (略)
 com/craftinginterpreters/lox/TokenType.class
$ java com.craftinginterpreters.lox.Lox doughnut.lox  3
CLASS class null
IDENTIFIER Doughnut null
LEFT_BRACE { null
   (略)
PRINT print null
STRING "Pipe fill of custrard and coast with chcolate." Pipe fill of custrard and coast with chcolate.
SEMICOLON ; null
RIGHT_BRACE } null
   (略)
RIGHT_PAREN ) null
SEMICOLON ; null
EOF  null
> ^C  4
```

1. `javac` コマンドを使って、作成したプログラムをコンパイルした。
2. `ls` コマンドで、classファイルができているか確認した。
3. `java` コマンドを使って、 jLox を起動した。`doughnut.lox` を指定した。
4. `Ctrl + C` （CtrlキーとCキーを同時に押して）インタプリタを停止した。

変更をコミット（commit）する

```
$ git status  1
$ git add .  2
$ git status  3
$ git commit -m "4章まで作成した"  4
$ git push  5
```

```
Username for 'https...（リポジトリのURL）': （GitHubアカウント名） 6
Password for 'https:...（リポジトリのURL）': （Personal access token） 7
```

1 変更点を確認した。

2 追加・更新したファイルをコミット対象に追加した。

3 もう一度Statusで追加できていることを確認した。

4 コミットした。

5 GitHub側のリポジトリへプッシュする。

6 ユーザー名を求められたら、 GitHubアカウント名を入力する。

7 パスワードを求められたら、パスワードの代わりにPersonal access tokenを入力する。

## 4.3. テキストの5章を作成する

1. Visual Studio Codeなどのテキストエディタで、5章までのコードを打ち込み、保存しなさい。
2. 次の実行例に従ってプログラムをコンパイルし、実行しなさい。

実行できるか確認する

```
$ javac com/craftinginterpreters/lox/AstPrinter.java 1
$ java com.craftinginterpreters.lox.AstPrinter 2
(* (- 123) (group 45.67))
```

1 `AstPrinter.java` をコンパイルした。

2 `AstPrinter` を実行した。

5章が終わったら、コミットしてプッシュしなさい。

## 4.4. テキストの6章を作成する

1. Visual Studio Codeなどのテキストエディタで、6章までのコードを打ち込み、保存しなさい。

2. 次の実行例に従ってプログラムをコンパイルし、実行しなさい。

実行できるか確認する

```
$ javac com/craftinginterpreters/lox/Lox.java 1
$ java com.craftinginterpreters.lox.Lox 2
> 4 * ( 5 + 3 )
(* 4.0 (group (+ 5.0 3.0))) 3
```

1 `Lox.java` をコンパイルした。

2 `Lox` を実行した。

3 Loxが、5章のAstPrinterと同じように動作するようになった。

6章が終わったら、コミットしてプッシュしなさい。

## 4.5. テキストの7章を作成する

1. Visual Studio Codeなどのテキストエディタで、7章までのコードを打ち込み、保存しなさい。
2. 次の実行例に従ってプログラムをコンパイルし、実行しなさい。

実行できるか確認する

```
$ javac com/craftinginterpreters/lox/Lox.java 1
$ java com.craftinginterpreters.lox.Lox 2
> 1 + 1 3
2
> 2 * ( 3 + 4 )
14
>
```

1 `Lox.java` をコンパイルした。

2 `Lox` を実行した。

3 Loxが、式を与えると計算結果を返すようになった。

7章が終わったら、コミットしてプッシュしなさい。

## 4.6. テキストの8章を作成する

1. Visual Studio Codeなどのテキストエディタで、8章までのコードを打ち込み、保存しなさい。
2. 次の実行例に従ってプログラムをコンパイルし、実行しなさい。

実行できるか確認する

```
$ javac com/craftinginterpreters/lox/Lox.java  1
$ java com.craftinginterpreters.lox.Lox block_test.lox  2
inner a  3
outer b
global c
outer a
outer b
global c
global a
global b
global c
```

1 `Lox.java` をコンパイルした。

2 `Lox` を使って `block_test.lox` を実行した。

3 Loxが、ブロックと変数のスコープを解釈できるようになった。

8章が終わったら、コミットしてプッシュしなさい。

## 4.7. テキストの9章を作成する

1. Visual Studio Codeなどのテキストエディタで、9章までのコードを打ち込み、保存しなさい。
2. 次の実行例に従ってプログラムをコンパイルし、実行しなさい。

実行できるか確認する

```
$ javac com/craftinginterpreters/lox/Lox.java  1
$ java com.craftinginterpreters.lox.Lox sample02.lox  2
60  3
false
CCCccc
true
0.5
```

1 `Lox.java` をコンパイルした。

2 `Lox` を使って `block_test.lox` を実行した。

3 Loxが、ブロックと変数のスコープを解釈できるようになった。

9章が終わったら、コミットしてプッシュしなさい。

# 5. 演習課題3

本講義の演習に使用したテキストのAmazonの商品ページのカスタマーレビュー欄に、このテキストを使って演習したことについてカスタマーレビューを投稿しなさい。

Amazonでは、Amazonで商品を購入していない場合（他社で購入など）についても、その商品のカスタマーレビューを投稿できるようになっている。 ただし、Amazonで購入した場合とは、掲載する数や掲載までの時間に違いがある。

**「インタプリタの作り方 －言語設計／開発の基本と2つの方式による実装－」**
https://www.amazon.co.jp/dp/4295017876?ref_=ppx_hzsearch_conn_dt_b_fed_asin_title_2

1. Amazonのウェブサイトにアクセスする。
   - https://www.amazon.co.jp/
2. Amazonにアカウントを持っていないなら、作成する。
3. ログインする。
4. 上記書籍のページを開く。

5. ページ後半の「カスタマーレビュー」の中の「カスタマーレビューを書く」をクリックして、カスタマーレビューのページを開く。
6. ☆をつけ、レビューを書く。適宜タイトルもつけること。
7. 投稿は、すぐに掲載されない。Amazonのチェックが済むと掲載される。
8. 書評が掲載されたら、それを本課題の成果とする。

## 6. 再評価演習の提出

このAssignmentについて、最終的な commit、pushの結果をもって再評価演習を提出したとみなす。

Last updated 2025-02-28 11:53:22 +0900