

# PRG1 (1): プログラム・コンパイラ・実行

脇田建

---

2016.9.26

# 講義のサポートサイト

---

❖ **<https://github.com/is-prg1a/lecture>**



# プログラムの実行方式

---

# プログラムの実行方式

---

- ❖ 直接実行方式（気合いで頑張る）
- ❖ コンパイラを用いた直接実行方式 (C, C++ など)
- ❖ インタプリタを用いた解釈実行方式 (JavaScript, Ruby, Python など)
- ❖ 仮想命令コンパイラと仮想機械を用いた解釈実行方式 (Java, Scala など)



# 直接実行方式

---

プログラム

機械命令の列

実行の主体

ハードウェア（中央演算装置;  
CPU）が機械語で用意されたプログラムを直接実行

計算

命令の読み込み、解釈、実行、  
プログラムカウンタの更新

# 簡単なプログラム例 (C)

---

```
int simple(int a, int n) {  
    for (int i = 1; i <= n; i++) {  
        a = a + i;  
    }  
    return a;  
}
```



# 機械命令

|           | 変数 |
|-----------|----|
| -4(%rbp)  | a  |
| -8(%rbp)  | n  |
| -12(%rbp) | i  |

| ラベル    | 命令   | 命令の引数           | 簡単な解説                  |
|--------|------|-----------------|------------------------|
| LBB0_1 | movl | -12(%rbp), %eax | eax := rbp[-12]        |
|        | cmpl | -8(%rbp), %eax  | rbp[-8] の値と eax の値を比較  |
|        | jg   | LBB0_4          | 比較結果が > なら LBB0_4 から実行 |
| BB#2   | movl | -12(%rbp), %eax | eax := rbp[-12]        |
|        | addl | -4(%rbp), %eax  | eax := eax + rbp[-4]   |
|        | movl | %eax, -4(%rbp)  | rbp[-4] = %eax         |
| BB#3   | movl | -12(%rbp), %eax | eax := rbp[-12]        |
|        | addl | \$1, %eax       | eax := eax + 1         |
|        | movl | %eax, -12(%rbp) | rbp[-12] := eax        |
|        | jmp  | LBB0_1          | 次はLBB0_1 から実行          |
| LBB0_4 | ...  | ...             |                        |



# 機械命令と変数の対応

|           | 変数 |
|-----------|----|
| -4(%rbp)  | a  |
| -8(%rbp)  | n  |
| -12(%rbp) | i  |

| ラベル    | 命令   | 命令の引数           | 簡単な解説                  |
|--------|------|-----------------|------------------------|
| LBB0_1 | movl | -12(%rbp), %eax | $A := i$               |
|        | cmpl | -8(%rbp), %eax  | n の値と A の値を比較          |
|        | jg   | LBB0_4          | $A > n$ なら LBB0_4 から実行 |
| BB#2   | movl | -12(%rbp), %eax | $A := i$               |
|        | addl | -4(%rbp), %eax  | $A := A + a$           |
|        | movl | %eax, -4(%rbp)  | $a = A$                |
| BB#3   | movl | -12(%rbp), %eax | $A := i$               |
|        | addl | \$1, %eax       | $A := A + 1$           |
|        | movl | %eax, -12(%rbp) | $i := A$               |
|        | jmp  | LBB0_1          | 次はLBB0_1 から実行          |
| LBB0_4 | ...  | ...             |                        |

つまり、 $a = i + a$

つまり、 $i = i + 1$



# 機械命令でのループ

|           | 変数 |
|-----------|----|
| -4(%rbp)  | a  |
| -8(%rbp)  | n  |
| -12(%rbp) | i  |

| ラベル    | 命令   | 命令の引数           | 簡単な解説                  |
|--------|------|-----------------|------------------------|
| LBB0_1 | movl | -12(%rbp), %eax | $A := i$               |
|        | cmpl | -8(%rbp), %eax  | n の値と A の値を比較          |
|        | jg   | LBB0_4          | $A > n$ なら LBB0_4 から実行 |
| BB#2   | movl | -12(%rbp), %eax | $A := i$               |
|        | addl | -4(%rbp), %eax  | $A := A + a$           |
|        | movl | %eax, -4(%rbp)  | $a = A$                |
| BB#3   | movl | -12(%rbp), %eax | $A := i$               |
|        | addl | \$1, %eax       | $A := A + 1$           |
|        | movl | %eax, -12(%rbp) | $i := A$               |
|        | jmp  | LBB0_1          | 次はLBB0_1 から実行          |
| LBB0_4 | ...  | ...             |                        |

つまり、 $a = i + a$

つまり、 $i = i + 1$



# 機械命令の直接実行

---

- ❖ CPUの性能の限界を引き出せる.
- ❖ コンパクトなコードを生成できる可能性がある.
- ❖ 猟奇的なプログラミングもやりやすい
- ❖ ↓ 機械命令は、難しい. デバッグが大変.



# コンパイラを用いた直接実行方式

---

- ❖ 動機：機械命令を人間が準備するのはあまりにつらい。助けて！
- ❖ コンパイラ：人間にとって理解し易いプログラミング言語（高級言語）を機械命令に翻訳するソフトウェア
  - ❖ 記述性が飛躍的に高まり，ソフトウェアの生産性が大幅に高まった
  - ❖ 最適化コンパイラの性能は年々向上しており，生成されたコードは十分な実行性能を誇る。
- ❖ 各種のコンパイラ
  - ❖ C: clang や gcc / C++: clang++ や g++ / OCaml: ocamlc



# コンパイラを用いた直接実行方式

---

\$ **cat simple.c**

```
#include <stdio.h>
```

```
int simple(int a, int n) {
```

```
    for (int i = 1; i <= n; i++) { a += i; }
```

```
    return a;
```

```
}
```

```
int main() { printf("1 + 2 + .. + 10 = %d\n", simple(0, 10)); }
```

プログラムを作成して

\$ **clang -o simple simple.c**

コンパイルして

\$ **./simple**

1 + 2 + .. + 10 = 55

実行



# コンパイラを用いる利点

```
int simple(int a, int n) {  
    for (int i = 1; i <= n; i++) {  
        a = a + i;  
    }  
    return a;  
}
```

```
movl    %edi, -4(%rbp)  
movl    %esi, -8(%rbp)  
movl    $1, -12(%rbp)  
LBB0_1:  
    movl    -12(%rbp), %eax  
    cmpl    -8(%rbp), %eax  
    jg      LBB0_4  
## BB#2:  
    movl    -12(%rbp), %eax  
    addl    -4(%rbp), %eax  
    movl    %eax, -4(%rbp)  
...
```

- ❖ メモリ配置 ( `-4(%rbp)` ) → 変数名 ( `a` )
- ❖ ラベル ( `LBB0_4` ) → 制御構造 ( **for** や **return** )
- ❖ 最適化コンパイラを利用するとかなりの実行性能が期待できる

# インタプリタを用いた実行方式

---

- ❖ 動機：「プログラム作成，コンパイル，実行」の繰り返しは面倒．すぐに実行したい．
- ❖ インタプリタ：プログラムを読み取り，意味を解釈しながら，その場で実行するプログラム
  - ❖ プログラム断片を徐々に入力し，少しずつ実行できるので，わかりやすい．
- ❖ 各種のインタプリタ
  - ❖ matlab, ocaml, perl, python, R, ruby, scala, Scheme (gauche, rabbit)



# インタプリタを用いた実行例

---

\$ **scala**

Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_25).

Type in expressions to have them evaluated.

Type :help for more information.

```
scala> def simple(a: Int, n: Int): Int = {  
      |   def aux(a: Int, i: Int): Int = {  
      |     if (i > n) a else aux(a + i, i + 1)  
      |   }  
      |     aux(a, 1);  
      | }
```

simple: (a: Int, n: Int)Int

```
scala> simple(0, 10)
```

res1: Int = 55

```
scala> simple(0, 30)
```

res2: Int = 465



# 参考：Scalaのインタプリタにファイルを読み込む方法

---

\$ **scala** Scalaのインタプリタの起動

Welcome to Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_25).

Type in expressions to have them evaluated.

Type :help for more information.

scala> **:load simple.scala**

Scalaファイル (simple.scala) の読み込み

Loading simple.scala...

defined object Simple

scala> **Simple.simple(0, 10)**

res0: Int = 55

simple.scala で定義されている

simple 関数の呼び出し



# インタプリタの働き

- ❖ 入力されたプログラムの構文解析

- ❖ 構文エラーの指摘

```
scala> (+ a 1)
<console>:1: error: ')' expected but
integer literal found.
(+ a 1)
    ^
```

- ❖ プログラムの意味の解釈

- ❖ 意味に関するエラーの指摘

```
scala> b + 1
<console>:11: error: not found:
value b
    b + 1
    ^
```

- ❖ プログラムの意味にしたがって評価・実行

- ❖ 実行結果の出力

```
scala> simple(0, 10)
res7: Int = 55
```



# 仮想機械を用いた実行方式

---

- ✧ バイトコードコンパイラ

- ✧ バイトコード（仮想命令） $\Leftrightarrow$  ネイティブコード（機械命令の別名）

- ✧ やや抽象度の高い命令（バイトコード）を準備する。

- ✧ 仮想機械：バイトコード列を解釈・実行する処理系

- ✧ バイトコードのインタプリタ（仮想CPUの真似をするソフトウェアと見做せる。  
ネイティブコンパイラでの実行より10～20倍遅い）

- ✧ JIT (Just-in-time) コンパイラ：バイトコード断片の仮想命令を解釈・実行しながら、対応するネイティブコードを蓄えておく。次に同じ断片が実行するときには、保存したネイティブコードを実行することで、高速に実行する。



# バイトコードコンパイラと仮想機械

| 言語                       | コンパイラ      | 仮想機械   | 基盤             |
|--------------------------|------------|--|----------------|
| Java                     | javac      | Java Virtual Machine (JVM)                   | あらゆる環境         |
| Scala                    | scalac     | Java Virtual Machine (JVM)                   | あらゆる環境         |
| C++ / C# / Basic / F# など | それぞれのコンパイラ | Microsoft 共通中間言語 (CIL)                       | .NET Framework |
| Java的な...                | javac      | Dalvik Virtual Machine<br>※最新のAndroidは使っていない | < Android 5.0  |



# 仮想機械を用いた実行例

\$ **cat simple.scala**

プログラムを作成

```
object Simple {  
  def simple(a: Int, n: Int): Int = {  
    def aux(a: Int, i: Int): Int = {  
      if (i > n) a else aux(a + i, i + 1)  
    }  
    aux(a, 1);  
  }  
  
  def main(arguments: Array[String]) {  
    println("1 + 2 + ... + 10 = " +  
      simple(0, 10))  
  }  
}
```

\$ **scalac simple.scala**

コンパイル: Simple.class, Simple\$.class

\$ **scala Simple**

1 + 2 + ... + 10 = 55

実行と結果の印字



# 仮想機械を用いる利点

---

- ❖ 可搬性 (portability): コンパイルしたプログラムを異なるアーキテクチャの機械で実行できる. OS (Android, iOS, Linux / OS X, UNIX, Windows, zOS) にもアーキテクチャ (ARM, AMD64, x86 / x64, POWER) にも依存しない実行方式.
- ❖ JITを利用すると, 思いの外, 高性能

# どうしたことでしょう

## Scalaのバイトコードは実は. . .

---

❖ `$ scalac simple.scala`

❖ `$ ls`

```
Makefile  Simple.class  simple.c  simple.scala
Simple$.class  simple  simple.s
```

❖ `$ file *.class`

```
Simple$.class: compiled Java class data, version 50.0 (Java 1.6)
Simple.class:  compiled Java class data, version 50.0 (Java 1.6)
```

❖ `java -cp /home/wakita/brew/opt/scala/libexec/lib/scala-library.jar:. Simple`

$1 + 2 + \dots + 10 = 55$



# バイトコードコンパイラと仮想機械

| 言語   | コンパイラ      | 仮想機械   | 基盤             |
|--|------------|--|----------------|
| Java   | javac      | Java Virtual Machine (JVM)                   | あらゆる環境         |
| Scala  | scalac     | Java Virtual Machine (JVM)                   | あらゆる環境         |
| VB .NET / C++ / C# / F# / J# / JScript / PowerShell など | それぞれのコンパイラ | Microsoft 共通中間言語 (CIL)                       | .NET Framework |
| Java的な...  | javac      | Dalvik Virtual Machine<br>※最新のAndroidは使っていない | < Android 5.0  |

# 多言語 → 共通仮想機械 (Microsoft)

## プログラミング言語

Visual  
Basic  
.NET

C++

C#

F#

J#

JScript

PowerShell

## コンパイル

## 中間言語

共通中間言語

(CIL; Common Intermediate  
Language)

## 仮想実行基盤

.NET Framework

## 実行基盤

演算装置 (Microprocessor)



# 多言語 → 共通仮想機械 (JVM)

## プログラミング言語

Java

Jython  
(JVM版 Python)

JRuby  
(JVM版 Ruby)

Scala

コンパイル

中間言語

Java Bytecode

仮想実行基盤

Java 仮想機械  
(JVM; Java Virtual Machine)

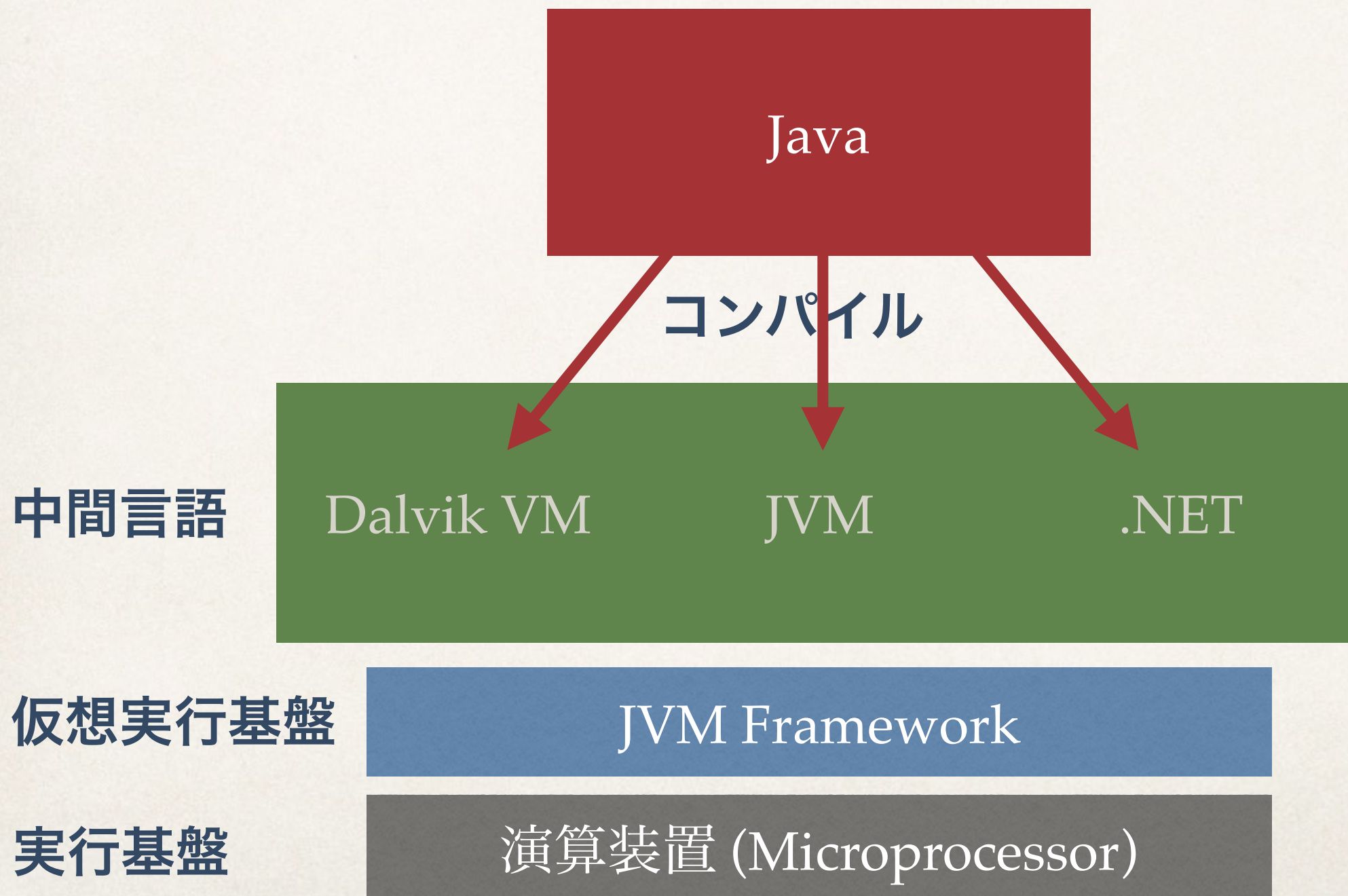
実行基盤

演算装置 (Microprocessor)



# 同一言語 → 多様な実行環境

プログラミング言語





# sbt – Scala の開発環境

---



# プログラムの開発ステップ

---

1. プログラムを書く
2. コンパイル：文法エラーや意味エラーに出会ったらステップ1へ
3. 実行：実行時エラーに出会ったらステップ1へ
4. テスト：テストに失敗したら、頭を冷してからステップ1へ
5. 完成！



# Scala開発の風景 – 各ステップでどんな作業をしているか考えて下さい

---

## ♣ scalac simple.scala

simple.scala:1: error: expected class or object definition

```
def simple(a: Int, n: Int): Int = {
```

^

simple.scala:8: error: expected class or object definition

```
def main(arguments: Array[String]) {
```

^

two errors found

## ♣ scalac simple.scala

simple.scala:4: warning: a pure expression does nothing in statement position; you may be omitting necessary parentheses

```
if (i > n) a else aux(a + i, i + 1)
```

^

simple.scala:6: error: type mismatch;  
found : Unit

required: Int

```
aux(a, 1);
```

^

one warning found

one error found

## ♣ scalac simple.scala

## ♣ scala Simple

1 + 2 + ... + 10 = 14

## ♣ scalac simple.scala

## ♣ scala Simple

1 + 2 + ... + 10 = 55



# Scala開発の風景 – 各ステップでどんな作業をしているか考えて下さい

---

- ❀ `scalac simple.scala`
- ❀ `scalac simple.scala`
- ❀ `scalac simple.scala`
- ❀ `scala Simple`
- ❀ `scalac simple.scala`
- ❀ `scala Simple`

プログラミング作業は、コンパイル、実行、テストの連続（たくさんタイピングしなくてはいけない）

しかも、scalac の実行は時間がかかる



# bashの便利な機能

---

- ❖ コマンド実行履歴機能：↑キー、↓キー
  - ❖ 以前実行したコマンドの再実行
- ❖ コマンド行編集機能：Ctrl-a, Ctrl-b (← キー), Ctrl-f (→ キー), Ctrl-e
  - ❖ コマンド入力中の小さな間違いを素早く修正するのに便利
- ❖ コマンド再実行：!
  - ❖ !! – 直前に実行したコマンドの再実行
  - ❖ !sc – コマンド実行履歴のなかで "sc" で始まるコマンドを探し、それを実行

# さらに便利な sbt (Scala build tool)

---

- ❖ Scala の開発環境（地味だけれど、とてもいい）
  - ❖ Scala プログラムのビルド（コンパイル & 統合）
  - ❖ 必要な Scala パッケージの自動インストール
  - ❖ 継続的コンパイル、継続的テスト
  - ❖ Scala インタプリタとの統合
  - ❖ Java のサポート
- ❖ 実はみなさんはすでに使っています。



# sbtの利用

---

gmac01:lx01 wakita\$ **sbt**

[info] Loading project definition from /home/wakita/classes/cs1/lx01/project

[info] Set current project to lecture (in build file:/home/wakita/classes/cs1/lx01/)

> **compile**

[info] Updating {file:/home/wakita/classes/cs1/lx01/}root...

[info] Resolving jline#jline;2.12.1 ...

[info] Done updating.

[info] Compiling 1 Scala source to /home/wakita/tmp/sbt/cs1g/lecture/scala-2.11/classes...

[success] Total time: 7 s, completed 2016/09/26 14:02:51

> **run**

[info] Running Simple

1 + 2 + ... + 10 = 55

[success] Total time: 0 s, completed 2016/09/26 14:02:54

> **exit**





sbtp

## sbt の起動

```
[info] Set current project to cs1-lx00a (in  
build file: /Users/wakita/tmp/lx00a/)
```

### run コマンドでコンパイル & 実行

```
> run
```

```
[info] Compiling 1 Scala source to /Users/  
wakita/tmp/cs1f/scala-2.11/classes...  
[error] /Users/wakita/tmp/lx00a/src/lx00-  
a.scala:2: expected class or object definition  
[error] def simple(a: Int, n: Int): Int = {  
[error] ^  
  
[error] /Users/wakita/tmp/lx00a/src/lx00-  
a.scala:9: expected class or object definition  
[error] def main(arguments: Array[String]) {  
[error] ^  
[error] two errors found  
[error] (compile:compileIncremental)  
Compilation failed  
[error] Total time: 2 s, completed 2015/10/06  
6:11:54
```

文法エラーだ！

### プログラムを修正して、再実行

```
> run
```

```
[info] Running Simple  
[info] 1 + 2 + ... + 10 = 14  
[success] Total time: 0 s, completed  
2015/10/06 6:12:03
```

### プログラムを修正して、再実行

```
> run
```

```
[info] Compiling 1 Scala source to /Users/  
wakita/tmp/cs1f/scala-2.11/classes...  
[info] Running Simple  
[info] 1 + 2 + ... + 10 = 55  
[success] Total time: 3 s, completed  
2015/10/06 6:12:15
```



# 不精者のための sbt

---

- ❖ "run" を何度も入力するのが面倒じゃ
- ❖ コマンド履歴（↑と↓）を使って下さい
- ❖ ↑と↓を入力するのも面倒じゃ
- ❖ なんとなくワガママ
- ❖ そういうアンタに  
"~run" コマンド



☛ sbt

sbt セッションを通して実行したのは最初の ~run コマンドだけ

[info] Set current project to cs1-lx00a (in build file: /Users/wakita/tmp/lx00a/)

> ~run

[info] Compiling 1 Scala source to /Users/wakita/tmp/cs1f/scala-2.11/classes...

[error] /Users/wakita/tmp/lx00a/src/lx00-a.scala:2: expected class or object definition

[error] def simple(a: Int, n: Int): Int = {

[error] ^

[error] /Users/wakita/tmp/lx00a/src/lx00-a.scala:9: expected class or object definition

[error] def main(arguments: Array[String]) {

[error] ^

[error] two errors found

[error] (compile:compileIncremental) Compilation failed

[error] Total time: 2 s, completed 2015/10/06 6:27:08

**1. Waiting for source changes... (press enter to interrupt)**

[info] Compiling 1 Scala source to /Users/wakita/tmp/cs1f/scala-2.11/classes...

[info] Running Simple

[info] 1 + 2 + ... + 10 = 14

[success] Total time: 3 s, completed 2015/10/06 6:27:19

**2. Waiting for source changes... (press enter to interrupt)**

[info] Compiling 1 Scala source to /Users/wakita/tmp/cs1f/scala-2.11/classes...

[info] Running Simple

[info] 1 + 2 + ... + 10 = 55

[success] Total time: 1 s, completed 2015/10/06 6:27:30

**3. Waiting for source changes... (press enter to interrupt)**

プログラムを修正すると  
自動的にコンパイル&実行

継続的実行をやめるときは  
enter



# sbt の利用方法

- ❖ 作業用のフォルダを準備する

- ❖ **build.sbt**

- プロジェクトの設定

- ❖ **src**

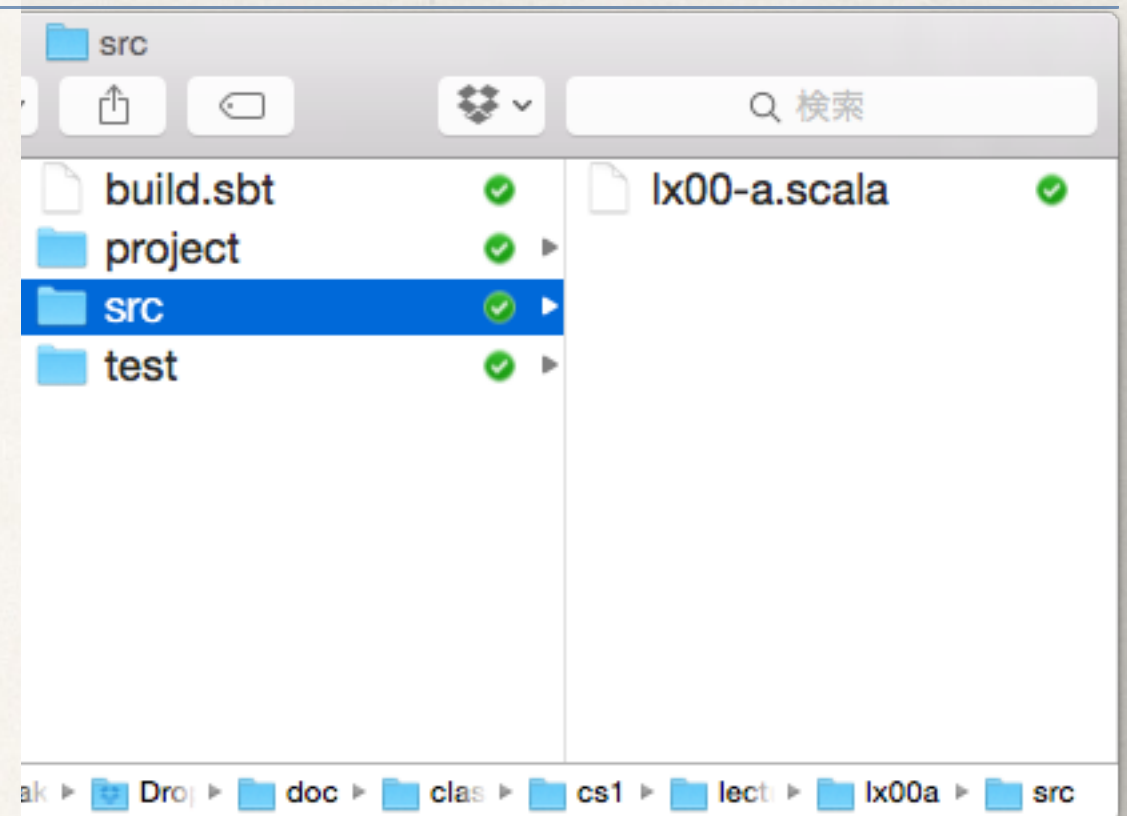
- プログラムを保存するディレクトリ

- ❖ **test**

- テストを保存するディレクトリ

- ❖ **project**

- sbtが勝手に作る



# 次回, 小テスト

---

- ❖ 本日の前半の内容（プログラムの実行方式）について復習して下さい