PRG1-w1a
本当のScalaプロジェクト

脇田建

2019.9.27

講義のサポートサイト

- **https://prg1-2019.github.io/lecture/**

次回のことはウェブサイトを参照

- **とても重要な内容**が書かれています。

  https://prg1-2019.github.io/lecture/web/w1a.html

今日の内容

- 本当の Scalaプロジェクト - コンパイラの利用
- 本当の Scala 開発 - sbt

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

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.
- プログラミングの超絶技巧を可能にする（コードを書き換えながら実行するとか)
- ↓でも、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！
- **ネイティブコードコンパイラ**：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．
- 各種のネイティブコードコンパイラ
  - C: **clang** や gcc / C++: clang++ や g++ / OCaml: ocamlopt

コンパイラを用いた直接実行方式

プログラムを作成して

```
$ 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)); }
```

コンパイルして

```
$ clang -o simple simple.c
```

実行

```
$ ./simple
1 + 2 + .. + 10 = 55
```

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 2.12.8 (Java
HotSpot(TM) 64-Bit Server VM, Java
11).
Type in expressions for evaluation. Or
try :help.
```

```
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)
res0: Int = 55

scala> simple(0, 30)
res1: 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 2.12.7 (Java HotSpot(TM) 64-Bit Server VM, Java 11).
Type in expressions for evaluation. Or try :help.

scala> :load simple.scala
Loading simple.scala...
defined object Simple

scala> Simple.simple(0, 10)
res0: Int = 55
```

Scalaファイル (simple.scala) の読み込み

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>:12: 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.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）⇔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 コンパイラ (Just-in-time compiler)**：バイトコード断片の仮想命令を解釈・実行しつつ，対応するネイティブコードを生成する．次回、同じ断片が実行するときは，そのネイティブコードを実行するので高速．

参考：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 ※最新のAndroidは使ってない	< 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

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 ※最新のAndroidは使ってない	< Android 5.0

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 のバイトコード

仮想実行基盤

Java 仮想機械

(JVM; Java Virtual Machine)

実行基盤

演算装置 (Microprocessor)

本当の Scala の開発環境 (sbt)

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 fix.scala

```
fix0.scala:1: error: expected class or object
definition
def simple(a: Int, n: Int): Int = {
^
fix0.scala:8: error: expected class or object
definition
println("1 + 2 + ... + 10 = " + simple(0, 10))
^
two errors found
```

- scalac fix.scala

```
fix1.scala:4: warning: a pure expression does
nothing in statement position
    if (i > n) a else aux(a + i, i + 1)
               ^
fix1.scala:6: error: type mismatch;
 found   : Unit
 required: Int
  aux(a, 1);
      ^
one warning found
one error found
```

- scalac fix.scala
- scala Fix

1 + 2 + ... + 10 = 11

バグった！

- scalac fix.scala
- scala Fix

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 fix.scala**
- **scalac fix.scala**
- **scalac fix.scala**
- **scala Fix**
- **scalac fix.scala**
- **scala Fix**

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 と scala は起動が遅い

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、でもとても便利）
  - 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: 起動

```
q3-prg$ sbt
[info] Loading settings for project global-plugins from plugins.sbt ...
[info] Loading global plugins from /Users/wakita/Dropbox/lib/sbt/1.0/plugins
[info] Loading project definition from /Users/wakita/Dropbox/doc/classes/q3-
prg/lecture/project
[info] Updating ProjectRef(uri("file:/Users/wakita/Dropbox/doc/classes/q3-prg/
lecture/project/"), "lecture-build")...
[info] Done updating.
[info] Loading settings for project root from build.sbt ...
[info] Set current project to root (in build file:/Users/wakita/Dropbox/doc/classes/
q3-prg/lecture/)
[info] sbt server started at local:///Users/wakita/.sbt/1.0/server/
317febbe5f3a3aa00801/sock

sbt:root>
```

sbtの利用:

プロジェクトlx01のコンパイル

```
sbt:root> lx01/compile
[info] Updating lx01...
[info] Done updating.
[info] Compiling 7 Scala sources to /Users/wakita/.tmp/sbt/prg19/
lx01/scala-2.12/classes ...
[warn] /Users/wakita/Dropbox/doc/classes/q3-prg/lecture/lx/lx01/
hello5a.scala:2:13: parameter value role in method hello is never used
[warn] いくつかの警告は省略

[success] Total time: 3 s, completed Sep 27, 2019, 9:55:57 AM
sbt:root> lx01/compile
[success] Total time: 0 s, completed Sep 27, 2019, 9:56:13 AM
sbt:root>
```

sbtの利用:

プロジェクトlx01の実行

```
sbt:root> lx01/run
[warn] Multiple main classes detected.  Run 'show discoveredMainClasses' to see the list
Multiple main classes detected, select one to run:
 [1] Hello1
 ...
 [7] Hello6
Enter number: [info] Packaging /Users/wakita/.tmp/sbt/prg19/lx01/scala-2.12/
lx01_2.12-0.1-SNAPSHOT.jar ...
[info] Done packaging.
1

[info] Running (fork) Hello1
[info] プログラミング第一へようこそ！

[success] Total time: 4 s, completed Sep 27, 2019, 9:58:47 AM

sbt:root>
```

sbtの利用:

プロジェクトlx01の指定

lx01/compile, lx01/run のようにプロジェクト名（ここではlx01）を指定するのが面倒な不精者のために

```
sbt:root> project lx01
[info] Set current project to lx01 (in build file:/Users/wakita/Dropbox/doc/classes/q3-prg/)

sbt:lx01> compile

sbt:lx01> run
```

sbt の起動

- **sbt**

lx01a_sbt プロジェクトを指定

sbt:root> **project lx01a_sbt**

run コマンドでコンパイル＆実行

sbt:x01a_sbt> **run**
[**error**] /Users/wakita/Dropbox/doc/classes/q3-prg/lx01a_sbt/src/fix.scala:1:1: expected class or object definition
[**error**] def simple(a: Int, n: Int): Int = {
[**error**] ^
…
[**error**] (Compile / compileIncremental)
Compilation failed

文法エラーだ！

プログラムを修正して、再実行

sbt:lx01a_sbt> **run**
[**error**] /Users/wakita/Dropbox/doc/classes/q3-prg/lx01a_sbt/src/fix.scala:6:8: type mismatch;
[**error**] found : Unit
[**error**] required: Int
[**error**] aux(a, 1);
[**error**] ^
…

型が間違ってる！

プログラムを修正して、再実行

sbt:x01a_sbt> **run**
[info] Running (fork) Fix
[info] 1 + 2 + ... + 10 = 11
12:24:10 PM

計算間違い

プログラムを修正して、再実行

sbt:x01a_sbt> **run**
[info] Compiling 1 Scala source to /Users/wakita/.tmp/sbt/prg18.is.titech.ac.jp/lx01a_sbt/scala-2.12/classes ...
[info] Running (fork) Fix
[info] 1 + 2 + ... + 10 = 55

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
- なんというワガママ
- 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

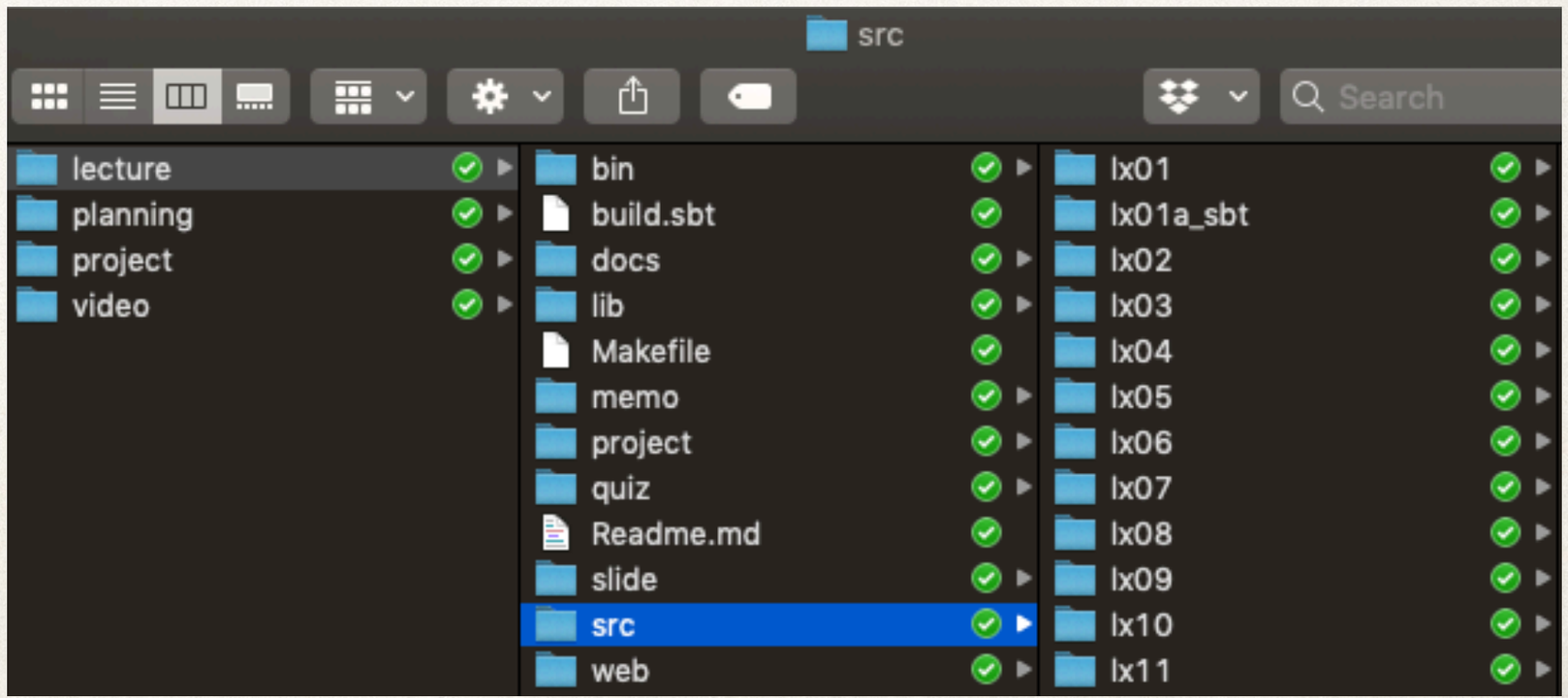
```
[info] Set current project to cs1-lx00a (in build file:/Users/wakita/tmp/lx00a/)
> ~run
[info] Compiling 1 Scala source to /Users/wakita/tmp/cs1f/scala-2.11/classes...
[error] /Users/wakita/tmp/lx00a/src/lx00-a.scala:2: expected class or object definition
[error]   def simple(a: Int, n: Int): Int = {
[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 プロジェクトの構成

- 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
  - **project**
    sbtが勝手に作る
  -



本当のScalaプログラム

本当の Scala プログラム

- Scala のプログラムの構造, **object 宣言**, **main**, **App**
  - **object / class / trait** さまざまな定義
    - **val x** = expr: 定数宣言
    - **var x** = expr: 変数宣言とその初期値の設定
    - **type x** = texpr: 型宣言
    - **def f(arguments ...): T => { ... }**: 関数やメソッドの定義

      特殊な関数 main

最小構成のプログラムの例

Hello1@GitHub ⇒

```
object Hello1 {
  def main(args: Array[String]) {
    println("プログラミング第一へようこそ！")
  }
}
```

最小構成のプログラムの例（別の流儀）

Hello2@GitHub ⇒

```
object Hello2 extends App {
  println("プログラミング第一へようこそ！")

}
```

main関数本体に**複数の文**を持つ場合

Hello3@GitHub ⇒

```
object Hello3 {
  def main(args: Array[String]) {
    println("プログラミング第一へようこそ！")
    println("講義担当は脇田 建（わきた けん）です。")
    println("演習担当は叢 悠悠（そう ゆうゆう）です。")
  }
}
```

定数宣言とその利用

Hello4@GitHub ⇒

```
object Hello4 extends App {
 val instructor1 = "脇田 建（わきた けん）"
  val instructor2 = "叢 悠悠（そう ゆうゆう）"
  println("講義の担当は" + instructor1 + "です。")
  println("演習の担当は" + instructor2 + "です。")
}
```

関数宣言とその利用

Hello5@GitHub ⇒

```
object Hello5 {
  def hello(role: String, professor: String) {
    println(role + "担当は" + professor + "です。")
  }
 def main(args: Array[String]) {
    hello("講義", "脇田 建（わきた けん）")

    hello("演習", "叢 悠悠（そう ゆうゆう）")

  }
}
```

変数宣言とその利用

Hello6@GitHub ⇒

```
object Hello6 extends App {
  val instructors = List("脇田 建（わきた けん）", "叢 悠悠（そう ゆうゆう）")

  var instr = 0
  def hello(role: String) {
    assert(instr < instructors.length)
    println(s"${role}担当は${instructors(instr)}です。")

    instr = instr + 1
  }
  hello("講義")

  hello("演習")

}
```

本日のまとめ

本日のウェブページ ⇒

- ガイダンス
- プログラムの実行方式
  - インタプリタとコンパイラ
- 本当のScala開発 (sbt)
- 本当のScala