

PRG1 (5): 状態を更新する機能

脇田建

2017.10.17

Designing Functions with Memory (HtDP 36)

記憶する機能の設計のステップ

- ❖ あなたのプログラムはなにかを**記憶**するのか？
- ❖ もし**記憶**するのなら
 - ❖ どのような内容を**記憶**するのか？
 - ❖ プログラムのどの機能がその**記憶**に関わるのか？

1. 記憶の必要性

- ❖ すべての人がプログラミングに熟達していたら記憶は不要かも
- ❖ 「記憶」を関数への引数として渡せばよいから
- ❖ でも、みんながScalaのインタプリタを利用できるわけじゃない

記憶機能が必要となる場合

1. システムが複数の機能を提供する場合（例：電話帳への追加と検索）
2. 提供する機能はひとつだが、状況や履歴に応じて振舞いに変化する場合（例：信号機の状態更新）

例 1：住所録（複数の機能）

- ❖ 住所録へのデータの追加の機能
- ❖ 住所録の検索の機能
- ❖ この場合の記憶はなに？
- ❖ この場合の機能はなに？

例 2 : クローク (複数の機能)

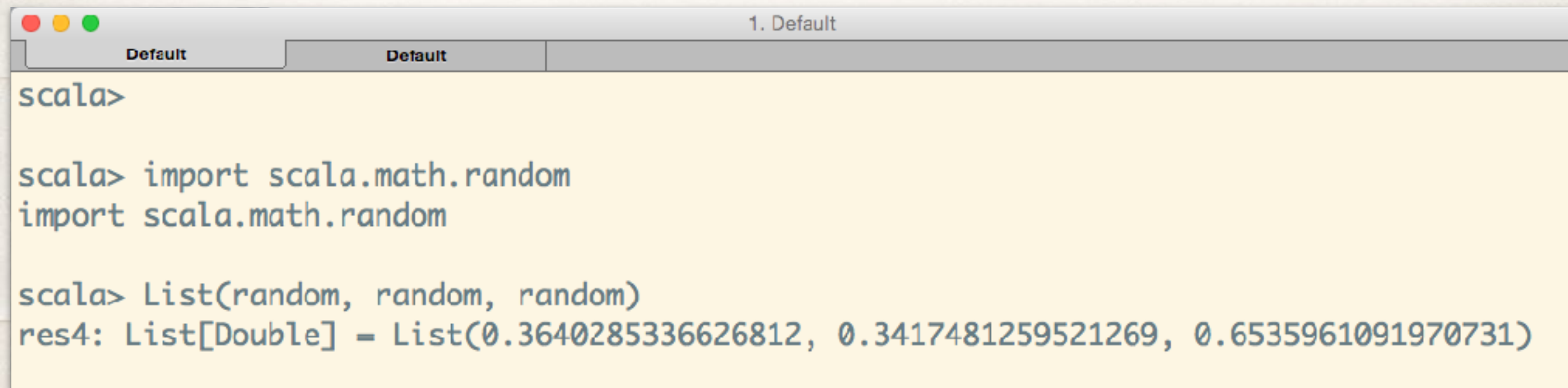
- ❖ 預入 : 荷物を渡すとタグを返す
- ❖ 返却 : タグを渡すと荷物を返す
- ❖ この場合の**記憶**はなに？
- ❖ この場合の**機能**はなに？

例3：信号機（履歴に応じた振舞い）

- ❖ 赤 → 青 → 黄 → 赤 → 青 → 黄 → 赤 → 青 → 黄 → ...
- ❖ この場合の**記憶**は？
- ❖ この場合の**機能**は？

例 4 : 乱数 (scala.math.random)

❖ random: Double \rightarrow [0, 1)



The screenshot shows a Scala REPL window with three tabs labeled 'Default'. The first tab is active. The prompt 'scala>' is shown. The user enters 'import scala.math.random' and 'import scala.math.random'. Then the user enters 'List(random, random, random)' and the result is 'res4: List[Double] = List(0.3640285336626812, 0.3417481259521269, 0.6535961091970731)'.

```
scala>  
  
scala> import scala.math.random  
import scala.math.random  
  
scala> List(random, random, random)  
res4: List[Double] = List(0.3640285336626812, 0.3417481259521269, 0.6535961091970731)
```

❖ この場合の**記憶**は？

❖ この場合の**機能**は？

2. 状態変数の同定

- ❖ 「記憶」は Scala の **可変変数** によって実装される.
この「可変変数」は *var* 宣言されたものを指している.
val 宣言された **定数** や関数の **引数** と混同しないこと.

In principle, a single variable is enough to implement all memory needs, ...

- ❖ 「変数がひとつあれば，十分」 (HtDP 36.2)
 - ❖ `var v1; var v2; ...` のかわりに `var state = (v1, v2, ...)` で代用できるから.
 - ❖ とはいえ，これは不自然. 自然に表現可能な変数を割り出すことが大切
- ❖ 一方で，無闇に多くの変数を導入することは，混乱のもと
- ❖ システムの状態をできるだけ**簡潔に説明**できるもの考えることが大切。いずれの極端もあまりよくない。

状態更新を伴う機能のデザインレシピ

- ❖ データ解析
- ❖ 契約 (Contract) / 入出力の働き (Purpose) / 副作用 (Effect)
- ❖ 雛形 (The Template)
- ❖ 例 (Program Examples) → テストケース
- ❖ 定義本体 (The Body)

住所録へのレシピの適用

状態更新を伴う機能のレシピ

データ解析

データ解析：住所録を表現する変数

- ✧ 住所録の内容
 - ✧ Adam: 014-1421-356
 - ✧ Eve: 017-3405-08
- ✧ こんな内容を表現するのに相応わしいデータ型は？

データ解析：住所録を表現する変数

- ❖ 住所録の内容
 - ❖ Adam: 014-1421-356
 - ❖ Eve: 017-3405-08
- ❖ こんな内容を表現するのに相応わしいデータ型は？
 - ❖ `List[Symbol, String]`

データ解析：住所録を表現する変数

- ❖ 住所録の内容
 - ❖ Adam: 014-1421-356
 - ❖ Eve: 017-3405-08
- ❖ こんな内容を表現するのに相応わしいデータ型は？
 - ❖ List[Symbol, String]
 - ❖ Map[Symbol, String]

データ解析：住所録の初期値は？

- ❖ `type AddressBook = List[Symbol, String]` のとき
- ❖ どんな初期値が適切か考えよう

状態更新を伴う機能のレシピ

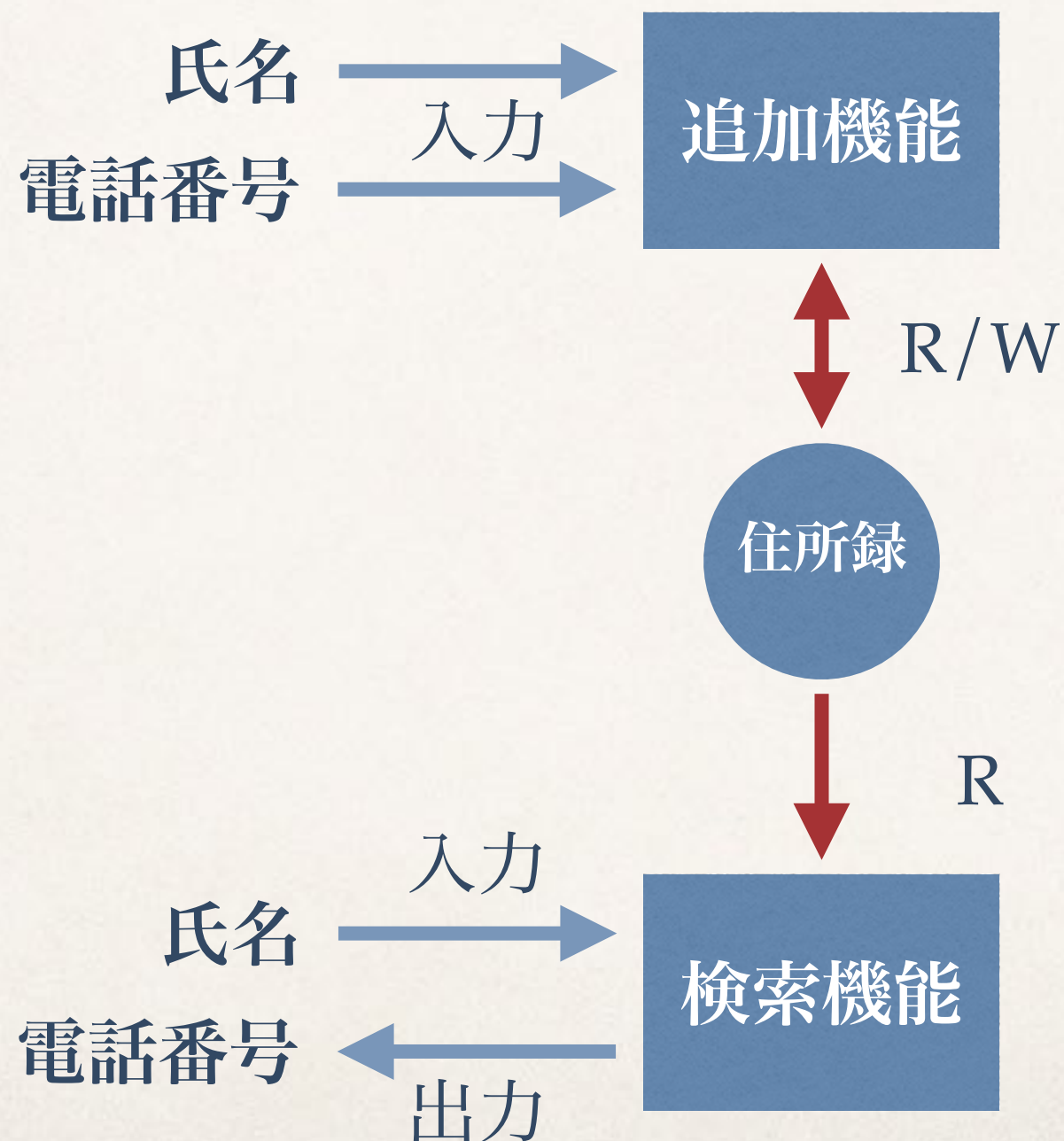
契約

契約：住所録へはどのような代入が考えられるか？

初期状態	[]
[]に (Adam, Adam's phone) を追加	[(Adam, Adam's phone)]
[(Adam, Adam's phone), ...] に (Eve, Eve's phone) を追加	[(Eve, Eve's phone), (Adam, Adam's phone), ...]
さらに(Adam, Adam's new phone) を追加	[(Adam, Adam's new phone), (Eve, Eve's phone), (Adam, Adam's phone), ...]

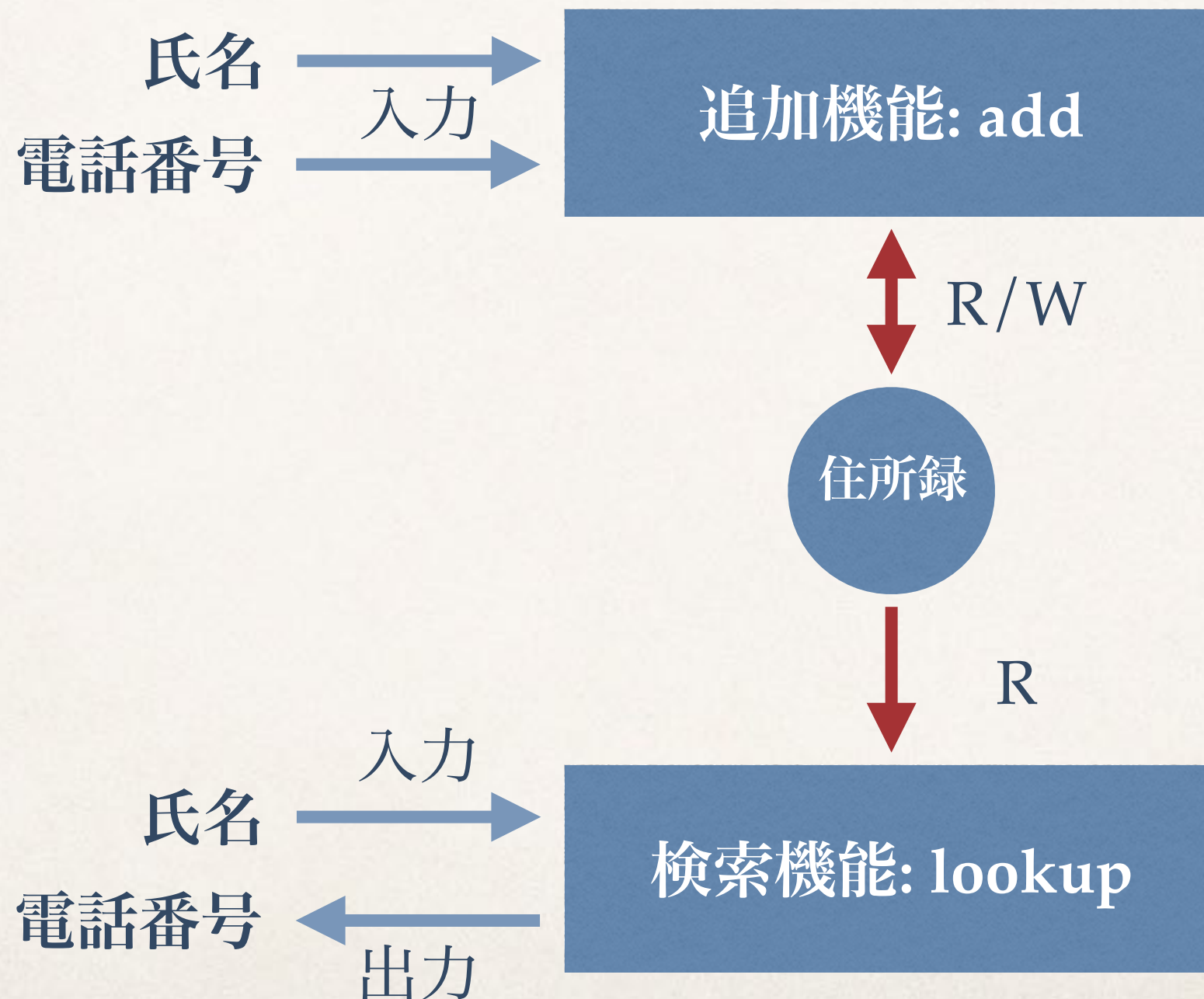
契約の図示

住所録を操作する機能の分析



契約の図示

住所録を操作する機能に命名



状態更新を伴う機能のレシピ プログラムの雛形へ

プログラムの雛形へ

概要

- ❖ 状態変数とその初期化
- ❖ 状態更新を伴わない機能の扱い
- ❖ 状態更新を伴う機能の扱い

プログラムの雛形へ

状態変数の宣言

- ❖ 状態変数の宣言とその初期値
 - ❖ 状態変数は契約の図中の変数ごとに
 - ❖ 初期値を与える関数を定義し、
 - ❖ `var` 宣言した状態変数を前述の初期値を与える関数で初期化する

状態変数の宣言

AddressBook.address_book

```
def initial_book() = List()
```

```
// address_book: 住所録を表現する状態変数. 初期状態では  
var address_book: AddressBook = initial_book()
```

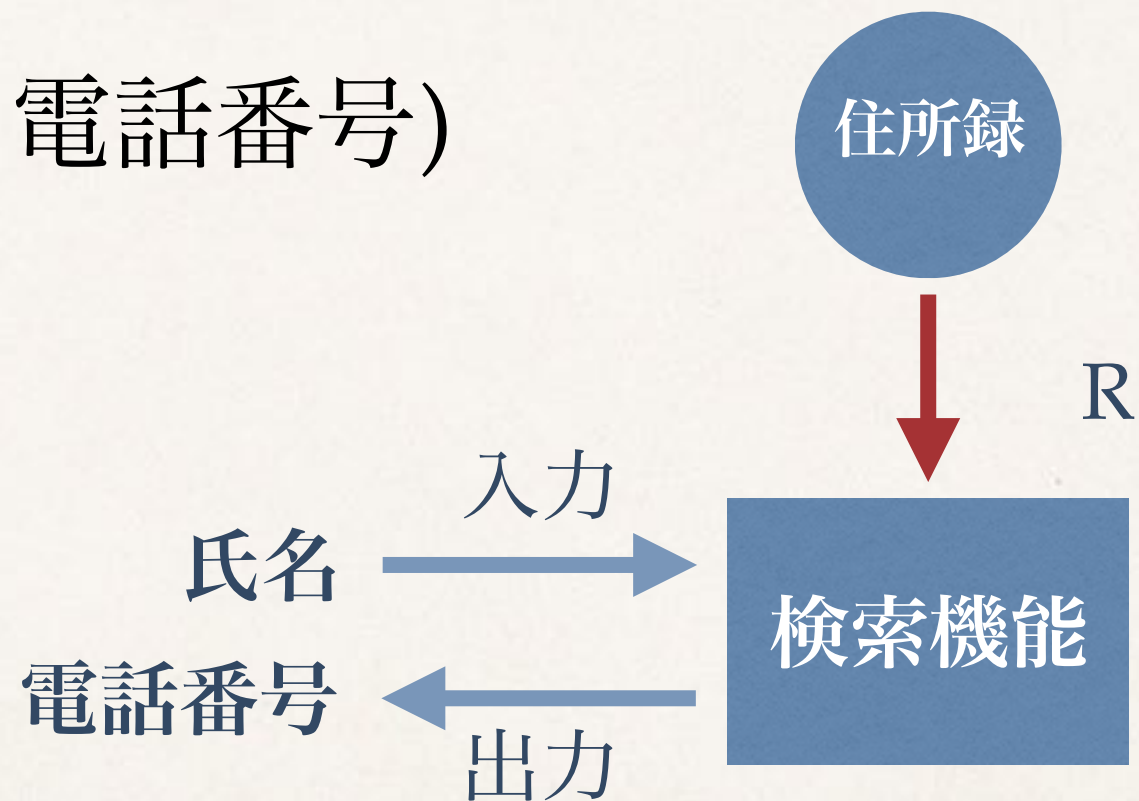

プログラムの雛形へ

状態更新を伴わない機能の場合

1. 契約の図中の各機能に沿って関数を定義し、契約を表現する。引数は契約の図に沿って与える。
2. 前述の関数を利用して、引数から定数と状態変数を除いたものを定義する。

AddressBook.lookup

- ❖ `def _lookup(住所録、氏名、電話番号)`
- ❖ `def lookup(氏名、電話番号)`



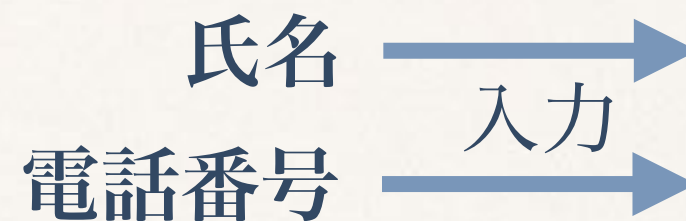
プログラムの雛形へ 状態更新を伴う機能の場合

1. 契約の図中の各機能に沿って関数を命名し、契約を表現する、**更新を伴わない関数**を定義する。引数は契約の図に沿って与える。
2. 前述の関数を利用して、**状態変数を更新するだけの関数**を定義する。

AddressBook.add

氏名
電話番号

入力



追加機能: add

R/W



住所録



❖ `def _add(住所録、氏名、電話番号)`

❖ `def add(氏名、電話番号)`

状態更新を伴う機能のレシピ

例 → テストケース

例 → テストケース

概要

- ❖ テストケースにおいては、状態更新を伴わない関数定義（名前が_で始まるもの）を用いたテストを記述する。

テストケースの例 (test/address-book.scala)

```
def add(address_book: AddressBook, data: (Name, PhoneNumber)): AddressBook =  
  AddressBook._add(address_book, data._1, data._2)
```

```
test("住所録の初期状態は空です") {  
  assert(initial_book == List())  
}
```

```
test("空の住所録にデータを追加したときに、住所録はそのデータを含むこと。") {  
  val address_book = add(initial_book, adam)  
  assert(_lookup(address_book, adam._1) == Some(adam._2))  
}
```

```
test("連続してふたつのデータを登録したら、どちらも登録されていること。") {  
  val address_book = add(add(initial_book, adam), eve)  
  assert(_lookup(address_book, adam._1) == Some(adam._2))  
  assert(_lookup(address_book, eve._1) == Some(eve._2))  
}
```

```
test("登録した情報を検索できること") {  
  val adam_added = add(initial_book, adam)  
  assert(_lookup(adam_added, adam._1) == Some(adam._2))  
  
  val eve_also_added = add(adam_added, eve)  
  assert(_lookup(eve_also_added, adam._1) == Some(adam._2))  
  assert(_lookup(eve_also_added, eve._1) == Some(eve._2))  
}
```

状態更新を伴う機能のレシピ

定義本体

AddressBook.lookup

```
/**
 * 関数 lookup
 * 契約: lookup: (Name) => 登録されているときは PhoneNumber、そうでない場合は空文字列
 *
 * 例: テストケース test/address-book.scalaを参照のこと
 */
def _lookup(address_book: AddressBook, name: Name): PhoneNumber = {
  for ((_name, phone) <- address_book) {
    if (_name == name) return phone
  }
  return ""
}

def lookup(name: Name) = _lookup(address_book, name)
```


AddressBook.add

```
/**
 * 関数 add
 *  契約: address_book に引数の対を追加したもの。
 *
 * 例: テストケース address-book.scalaを参照のこと
 */

def _add(address_book: AddressBook, name: Name, num: PhoneNumber): AddressBook =
  (name, num) :: address_book

def add(name: Name, num: PhoneNumber) { address_book = _add(address_book, name, num) }
```

信号機についてのデータ解析

信号機の色を表現する変数

- ❖ 信号の状態

- ❖ 赤 → 緑 → 黄 → ...

- ❖ ある時点での信号: 赤 or 緑 or 黄

- ❖ こんな内容を表現するのに相応わしいデータ型は？

信号機の色を表現する変数

- ✧ 信号の状態

- ✧ 赤 → 緑 → 黄 → ...

- ✧ ある時点での信号: 赤 or 緑 or 黄

- ✧ こんな内容を表現するのに相応わしいデータ型は？

- ✧ abstract class Color

- ✧ case class Red() extends Color

- ✧ case class Green() extends Color

- ✧ case class Yellow() extends Color

信号機の色を表現する変数

- ✧ 信号の状態

- ✧ 赤 → 緑 → 黄 → ...

- ✧ ある時点での信号: 赤 or 緑 or 黄

- ✧ こんな内容を表現するのに相応わしいデータ型は？

- ✧ `trait TrafficLight`

- ✧ `object Green extends TrafficLight`

- ✧ `object Yellow extends TrafficLight`

- ✧ `object Red extends TrafficLight`

信号機の色を表現する変数

- ✧ 信号の状態

- ✧ 赤 → 緑 → 黄 → ...

- ✧ ある時点での信号: 赤 or 緑 or 黄

- ✧ こんな内容を表現するのに相応わしいデータ型は？

- ✧ `trait TrafficLight`

- ✧ `Green = 0`

- ✧ `Yellow = 1`

- ✧ `Red = 2`

信号機の色の初期値は？

- ❖ 安全な初期値の原則

“In setting current_color to 'red, we follow a conventional rule of engineering to put devices into their least harmful state when starting it up.”

–HtDP 36.3 Functions that initialize memory

信号機ではどのような代入が考えられるか？

状態	次の状態
初期状態	赤
赤	緑
緑	黄
黄	赤

信号機を操作する代入の分析は？

信号機を表す変数への代入

- ❖ `current_color = Red()`
- ❖ `current_color = Green()`
- ❖ `current_color = Yellow()`

宿題

- ❖ 電話帳の例を参考にして、信号機の例について「状態更新を伴う機能のデザインレシピ」を適用しA4サイズのレポートとして提出すること。レポートの内容には少なくとも以下を含めること。
 - ❖ 信号機の例における契約を図示したもの（手書きで構わない）
 - ❖ 契約から作成したプログラムの雛形
 - ❖ テストケース
 - ❖ テストをすべてパスするプログラム
 - ❖ なお、1x05 のプログラムはデザインレシピに沿って作られていないので参考にしないこと。
- ❖ レポートの回収：次回の授業開始時

状態更新を伴う機能のデザインレシピ

- ❖ データ解析
- ❖ 契約 (Contract) / 入出力の働き (Purpose) / 副作用 (Effect)
- ❖ 雛形 (The Template)
- ❖ 例 (Program Examples) → テストケース
- ❖ 定義本体 (The Body)

4つの例題：信号の色の表現の違い

- ❖ signal1.scala: case class
- ❖ signal2.scala: trait に属する object
- ❖ signal3.scala: 数値で識別
- ❖ signal4.scala: ScalaFXに組み込みの Color オブジェクト