Guraプログラミング言語の紹介

www.gura-lang.org

LL Diver on Aug 23, 2014

自己紹介

名前	斉藤 寛 (ゆたか)
開発経験	組込ファームウェア から GUI まで
使用言語	C++, Gura
職歴	電機メーカ・米半導体メーカ・ベンチャ
現職	フリー というか無職

Gura 開発に専念中

Agenda

Guraとはなにか

基本的な仕様

イテレータ処理

拡張モジュール

Guraとはなにか

プログラム中で頻繁に出てくる 繰返し処理

```
for (i = 0; i < 10; i++) {
}
```

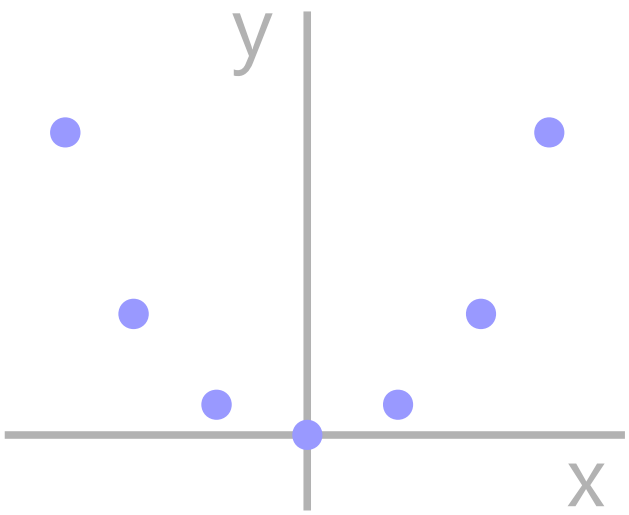
```
people.each do |per
end
```

```
for x in range(10):
    hoge
```

冗長な制御構文なしで処理できないか?

ケーススタディ (1)

数値列 -3, -2, -1, 0, 1, 2, 3 があります。
二乗した値を求めてリストにしてください



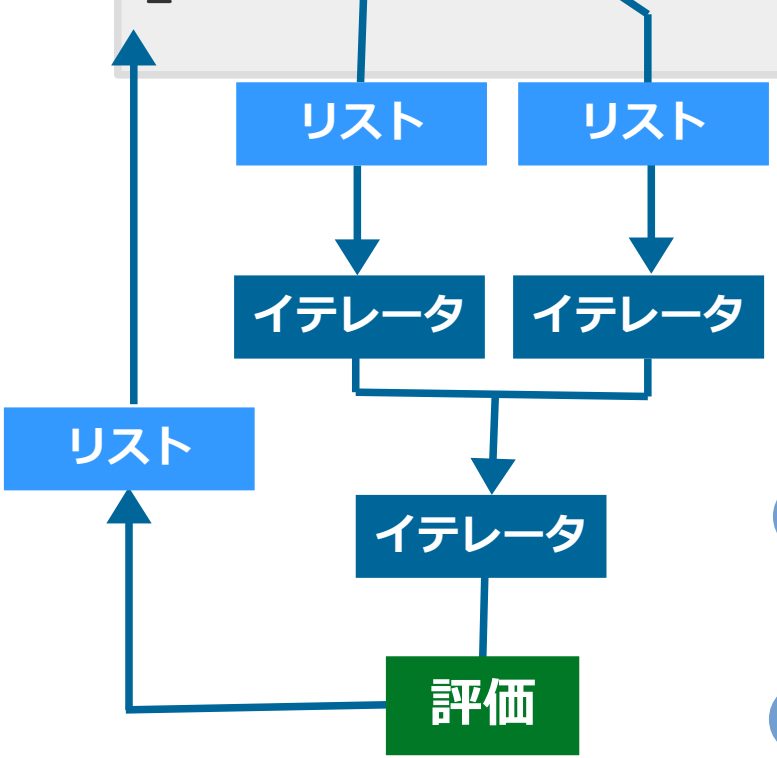
あなたの好きな言語で考えて …

Gura ならこう書く (1)

```
x = [-3, -2, -1, 0, 1, 2, 3]
y = x * x
```

Gura ならこう書く (1)

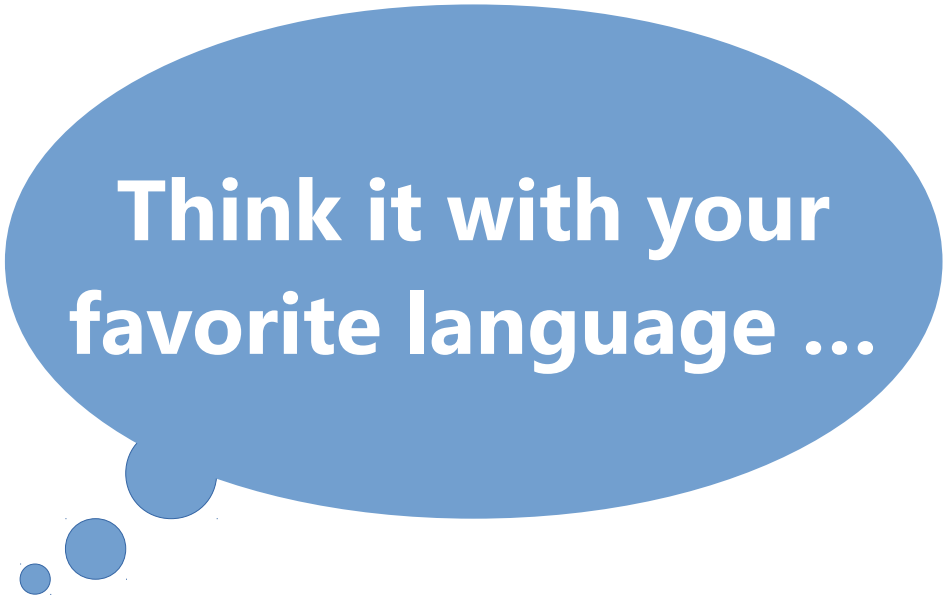
```
x = [-3, -2, -1, 0, 1, 2, 3]
y = x * x
```



1. リストからイテレータ生成
2. 要素をかけ算するイテレータ生成
3. イテレータを評価してリスト生成

ケーススタディ (2)

テキストファイルを読み込み、行番号つきで
画面に表示するプログラムをつくってください



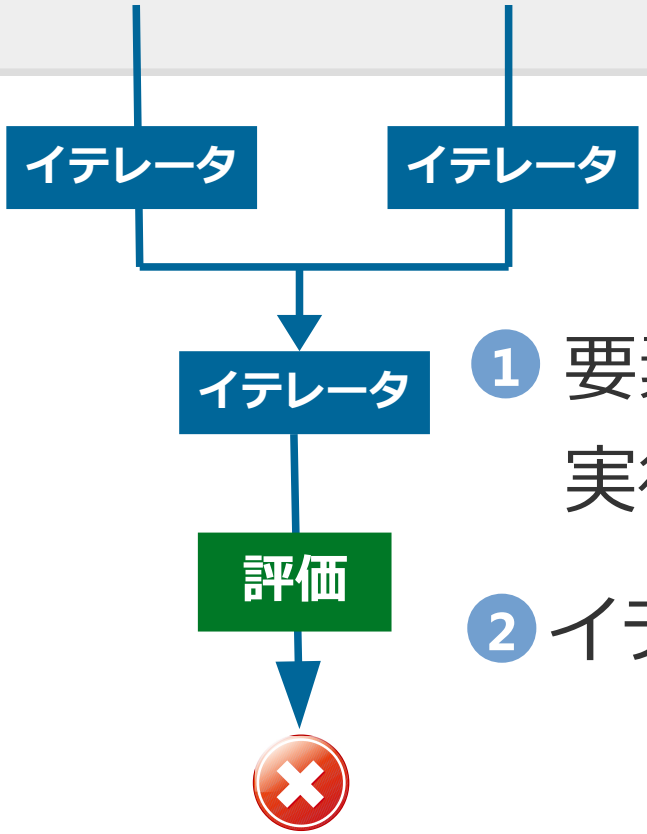

```
1: #include <std
2: int main()
3: {
4:     printf("H
5: }
```

Gura ならこう書く (2)

```
printf('%d: %s',
       1.., readlines('hello.c'))
```

Gura ならこう書く (2)

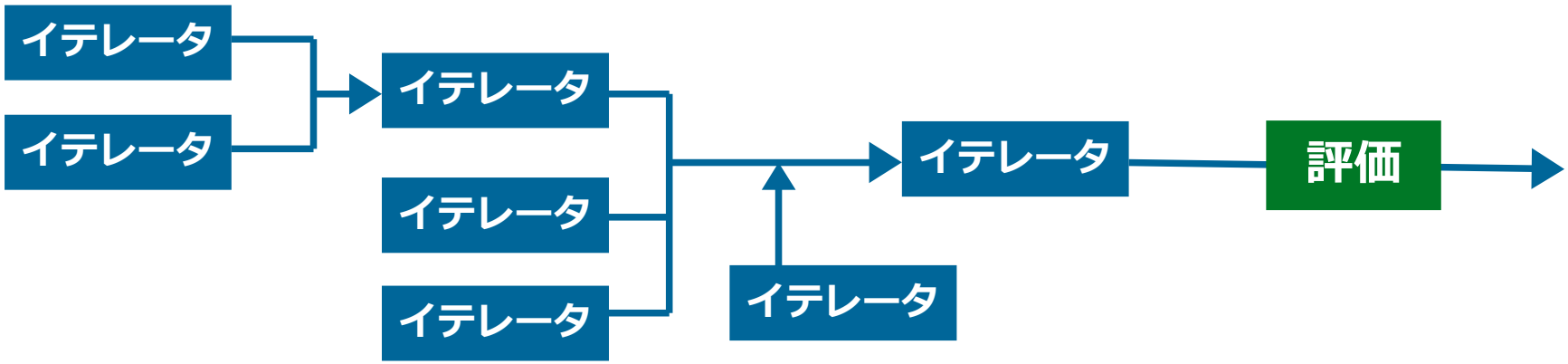
```
printf('%d: %s',
       1.., readlines('hello.c'))
```



1. 要素ごとに関数 `printf` を実行するイテレータ生成
2. イテレータを評価した後、破棄

つまり Gura とは

イテレータから新たなイテレータを生成し、演算・評価できる言語

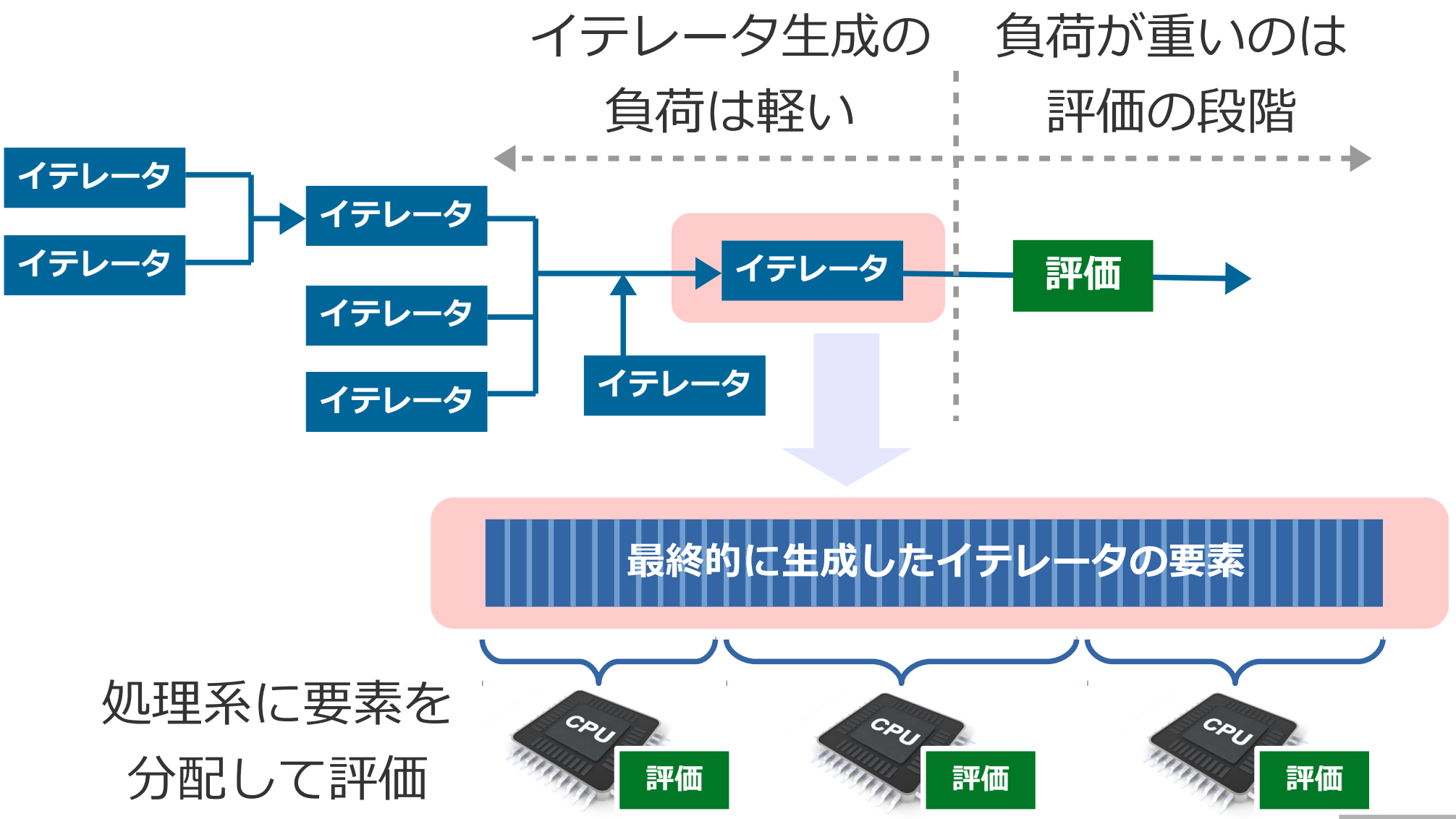


Guraにおいて、この操作を「マッピング」と呼ぶ

期待できること

1. 繰り返し処理を簡潔に書ける
2. 並列処理が容易になる かも

並列処理のアイデア



処理系に要素を
分配して評価

Agenda

Guraとはなにか

基本的な仕様

イテレータ処理

拡張モジュール

基本的な仕様

- 関数
- 制御構文
- オブジェクト指向
- コレクション
- スコープ管理

基本的な仕様(1) 関数

関数定義 (1)

```
f(a:number, b:number) = {
    a * a + b * b
}
```

引数の型を指定できる

呼出し例

```
x = f(3, 4)
```

```
x = f(a => 3, b => 4)
```

名前つき引数指定

基本的な仕様(1) 関数

関数定義 (2)

0 個以上の値をとる可変長引数
b+ なら 1 個以上の指定になる

```
f(a, b*) = {
    // any job
}
```

呼出し例

```
f(3)           // a=3, b=[]
f(3, 1)        // a=3, b=[1]
f(3, 1, 4, 1) // a=3, b=[1,4,1]
```

基本的な仕様(1) 関数

関数定義 (3)

```
my_loop(n) {block} = {
    while (n > 0) {
        block()
        n -= 1
    }
}
```

ブロック式を関数オブジェクトで受取る
{block?} ならオプショナルなブロック

呼出し例

```
my_loop(3) {
    println('hello')
}
```

基本的な仕様(2) 制御構文

繰返し処理

```
for (…) {
}
```

```
repeat (…) {
}
```

```
while (…) {
}
```

```
cross (…) {
}
```

条件分岐

```
if (…) {
} elsif (…) {
} elsif (…) {
} else {
}
```

例外処理

```
try {
} catch (…) {
} catch (…) {
}
```

基本的な仕様(3) オブジェクト指向

クラス定義

コンストラクタ

```
Fruit = class {
      __init__(name:string, price:number) = {
            this.name = name
            this.price = price
      }
      Print() = {
          printf('%s %d\n', this.name, this.price)
      }
}
```

インスタンス生成・メソッド呼出し

```
fruit = Fruit('Orange', 90)
fruit.Print()
```

基本的な仕様(3) オブジェクト指向

継承

```
A = class {
      __init__(x, y) = {
          // any jobs
      }
}

B = class(A) {
      __init__(x, y, z) = {|x, y|
          // any jobs
      }
}
```

ベースクラス
コンストラクタへの引数

基本的な仕様(4) コレクション

リスト

```
a = [3, 1, 4, 1, 5, 9]
b = ['zero', 'one', 2, 3, 'four', 5]
```

辞書

```
c = %{ `a => 3, `b => 1, `c => 4 }
d = %{
    'いぬ' => 'dog', 'ねこ' => 'cat'
}
```

基本的な仕様(5) スコープ管理

関数内はレキシカルスコープ

クロージャ

```
create_counter(n:number) = {
    function {
        n -= 1
    }
}
```

```
c = create_counter(4)
c()  // returns 3
c()  // returns 2
c()  // returns 1
```

Agenda

Guraとはなにか

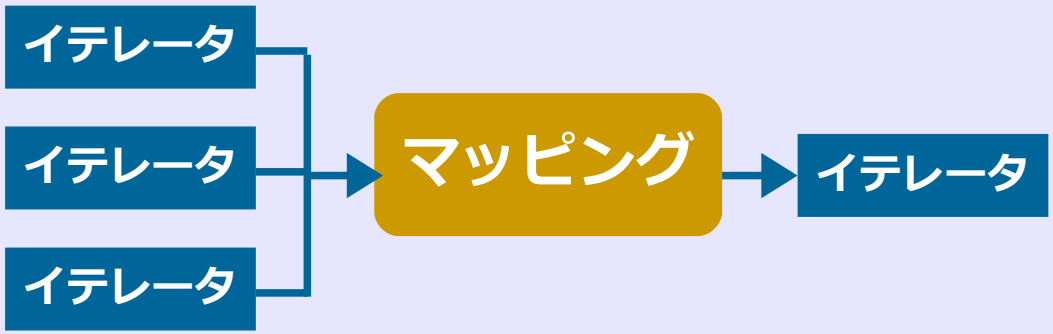
基本的な仕様

イテレータ処理

拡張モジュール

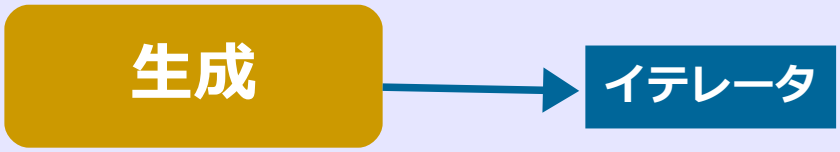
イテレータ処理

イテレータ処理: マッピングと生成



暗黙的マッピング

メンバマッピング



関数

繰返し制御構文

Gura における リストとイテレータ

リスト 要素がすべてメモリ上に用意されている

```
['apple', 'orange', 'grape']
```

ランダムアクセスが可能

イテレータ 要素をひとつずつ生成する

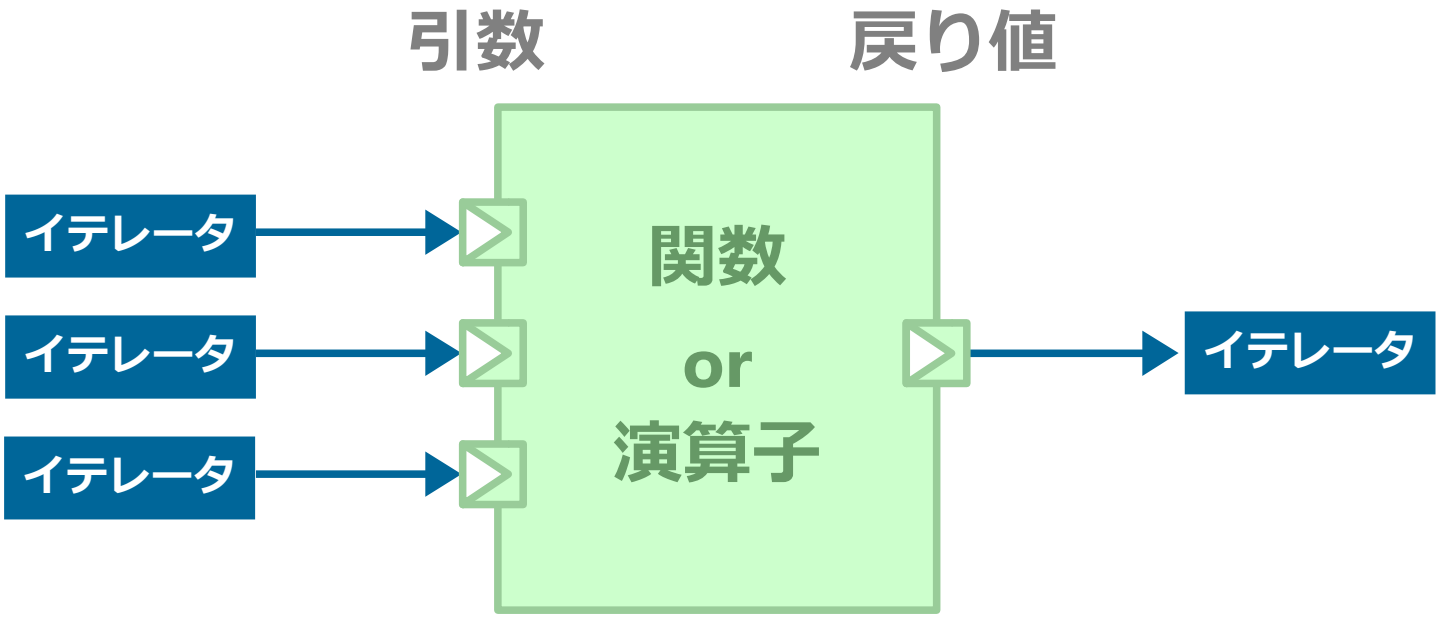
```
('apple', 'orange', 'grape')
```

評価しないと次の要素は分からない

イテレータ処理(1) 暗黙的マッピング

暗黙的マッピング

関数または演算子操作を行うイテレータを生成



イテレータ処理(1) 暗黙的マッピング

普通の関数

```
f(a:number, b:number) = {
    a * b
}
```

マッピング対応関数

アトリビュート :map をつける

```
f(a:number, b:number):map = {
    a * b
}
```

イテレータ処理(1) 暗黙的マッピング

数値

```
f(3, 4)
```

答: 12

リスト

```
f([2, 3, 4], [3, 4, 5])
```

答: [6, 12, 20]

イテレータ

```
f((2, 3, 4), (3, 4, 5))
```

答: (6, 12, 20)

数値とイテレータ

```
f(5, (3, 4, 5))
```

答: (15, 20, 25)

イテレータ処理(1) 暗黙的マッピング

マッピング適用は引数のデータ型によって異なる

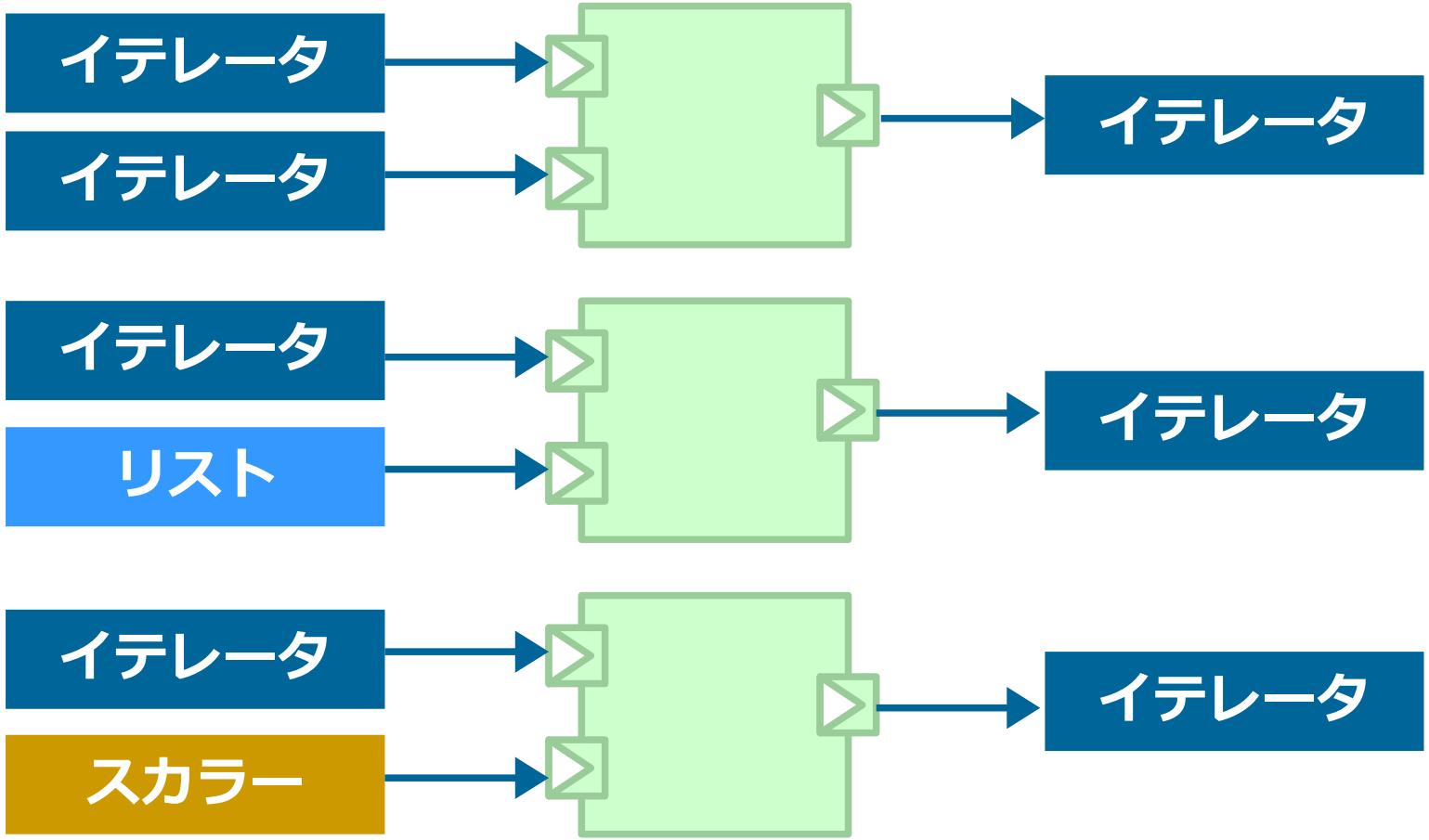
データ型を大別

- **イテレータ** イテレータ
- **リスト** リスト
- **スカラー** イテレータ・リスト以外

マッピング適用 3 つのルール

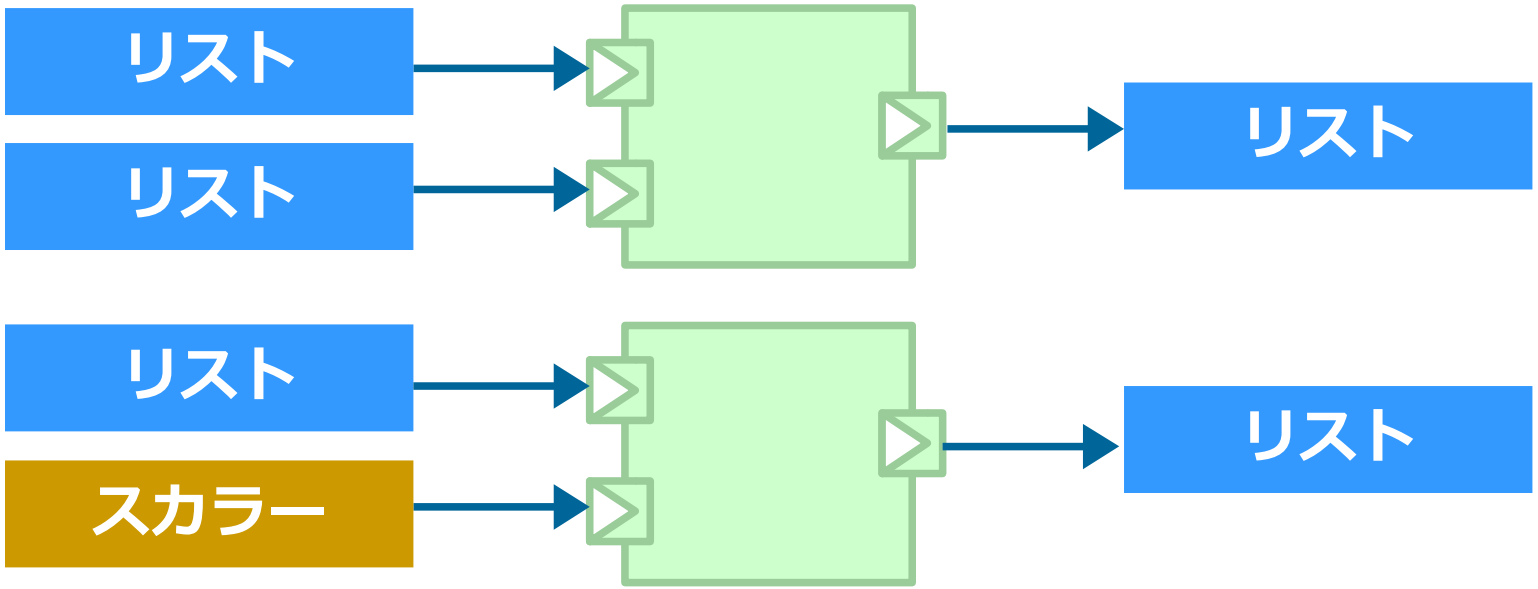
イテレータ処理(1) 暗黙的マッピング

ルール 1 引数にイテレータがあればイテレータを生成



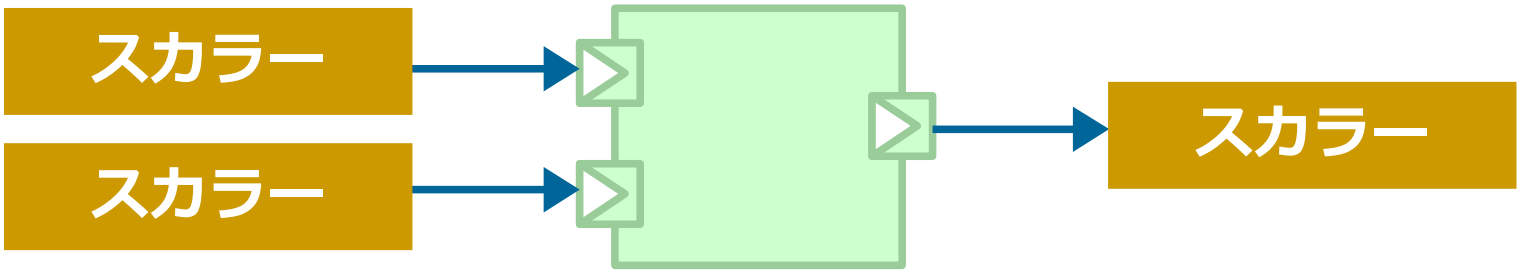
イテレータ処理(1) 暗黙的マッピング

ルール 2 引数にイテレータがなく、
リストがあればリストを生成



イテレータ処理(1) 暗黙的マッピング

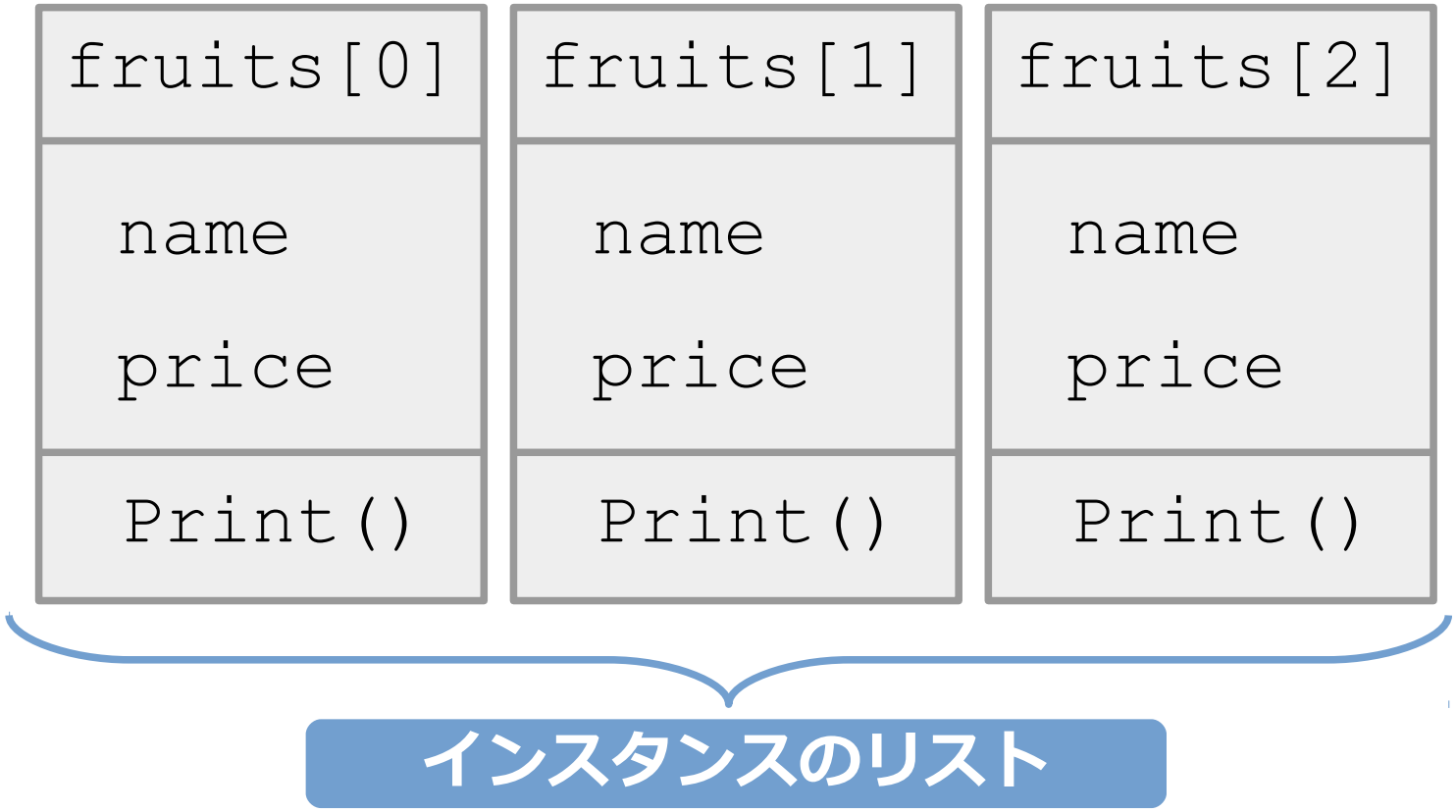
ルール 3 引数にスカラーのみがあればスカラーを生成



イテレータ処理(2) メンバマッピング

メンバマッピング

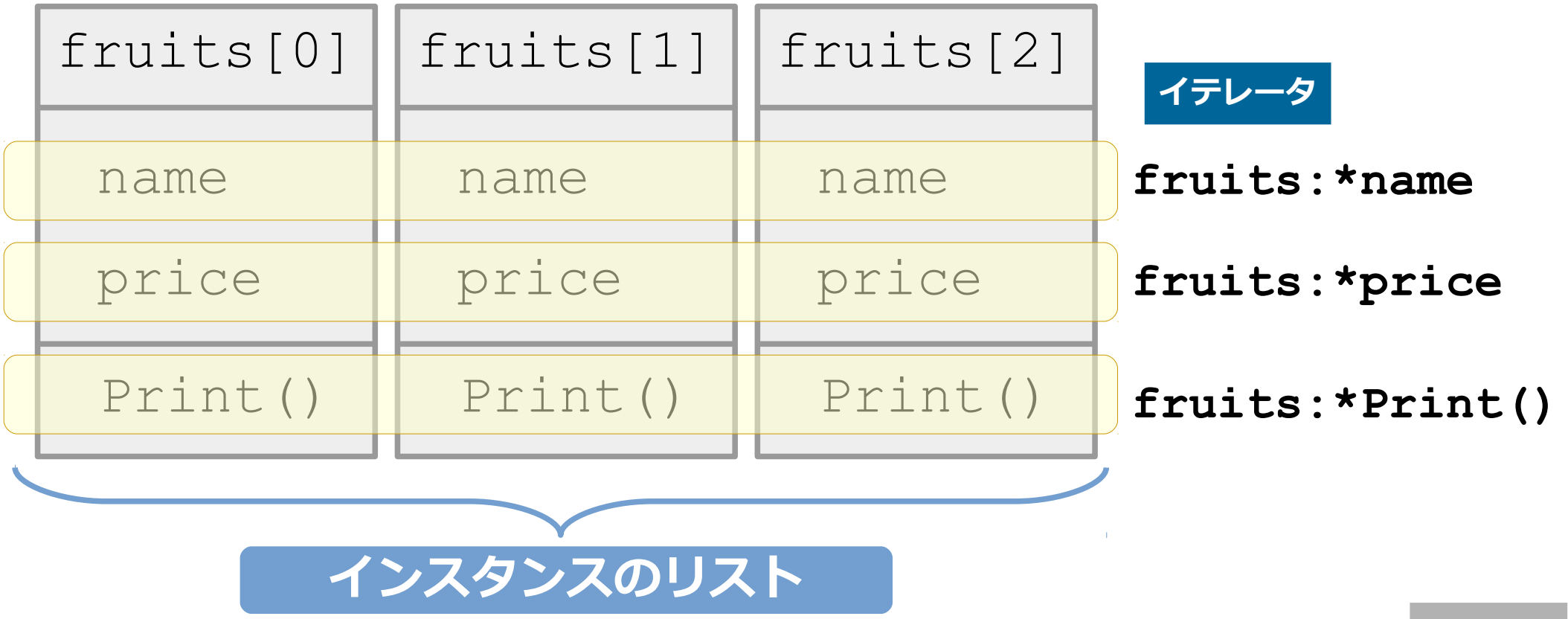
メンバアクセスをするイテレータを生成する機能



イテレータ処理(2) メンバマッピング

メンバマッピング

メンバアクセスをするイテレータを生成する機能



イテレータ処理(2) メンバマッピング

例題 `Fruit` インスタンスのメンバ `price` の合計を表示する

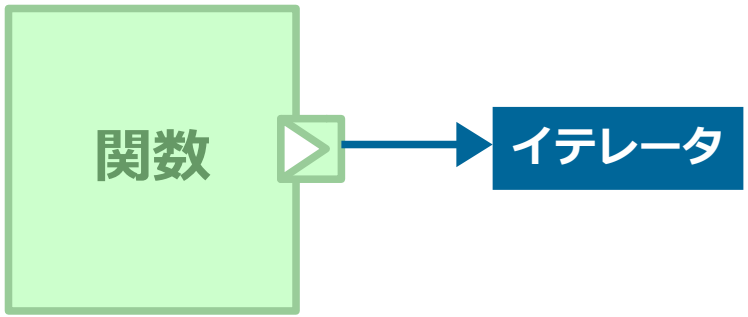
解法1 繰返し構文を使う

```
sum = 0
for (fruit in fruits) {
    sum += fruit.price
}
println(sum)
```

解法2 メンバマッピングを使う

```
println(fruits:*price.sum())
```

イテレータ処理(3) 関数



設計ポリシー

関数がデータ列を返す場合は
リストではなくイテレータで返す

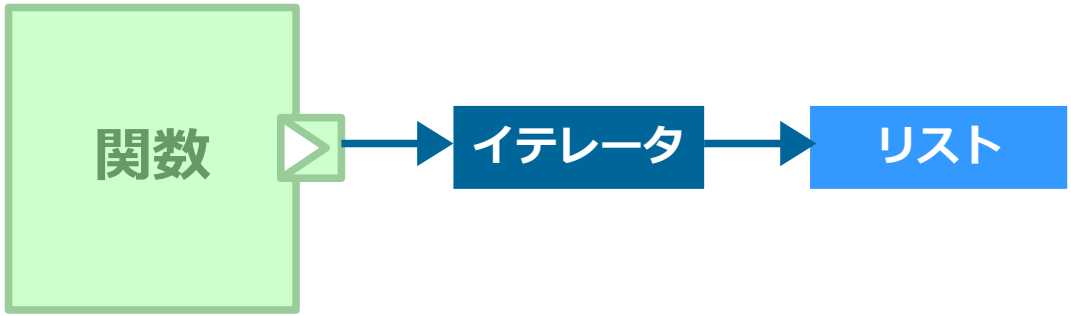
```
rtn = readlines('hello.c')
```

行ごとの文字列を返すイテレータ

```
rtn = range(10)
```

0 から 9 までの数値を返すイテレータ

イテレータ処理(3) 関数



リストがほしいときは
アトリビュート `:list` を
つけて呼び出し

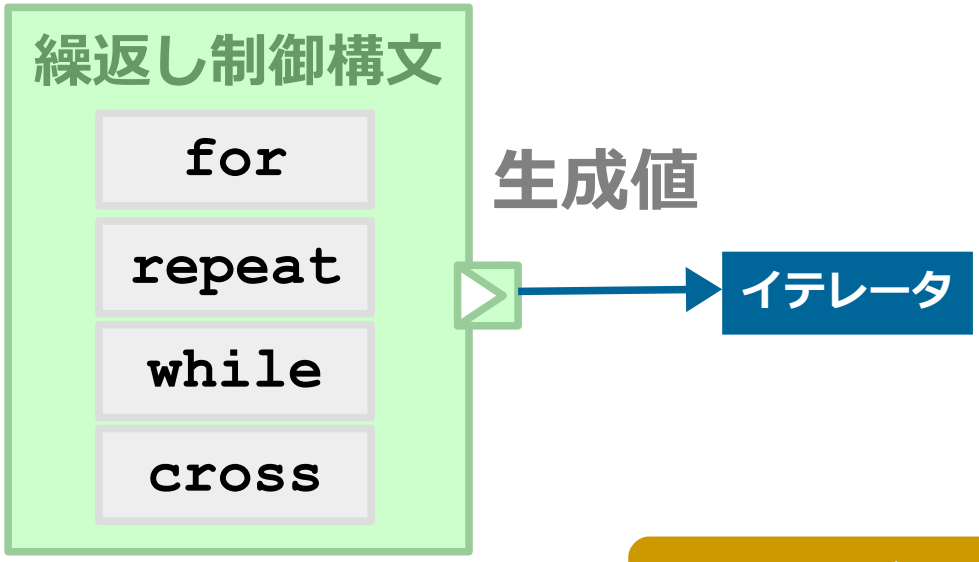
```
rtn = readlines('hello.c'):list
```

行ごとの文字列を含むリスト

```
rtn = range(10):list
```

0 から 9 までの数値を含むリスト

イテレータ処理(4) 繰返し制御構文



繰返し制御構文から
イテレータを生成

アトリビュート :iter をつける

```
x = for (...):iter {
    繰返し処理
}
```

繰返し処理の評価値が
イテレータの要素値になる

イテレータ処理(4) 制御構文イテレータ

制御構文イテレータの使用例

```
n = 0
x = for (i in 0..5):iter {
    n += i
}
```

この時点では何も実行しない

```
println(x)
```

結果を表示: 0 1 3 6 10 15

イテレータ処理(4) 制御構文イテレータ

素数を生成するイテレータ

```
prime() = {
    p = []
    for (n in 2..):xiter {
        if (!(n % p.each() == 0).or()) {
            p.add(n)
            n
        }
    }
}
```

```
primes = prime()
```

素数(2, 3, 5, 7..)を返すイテレータ

Agenda

Guraとはなにか

基本的な仕様

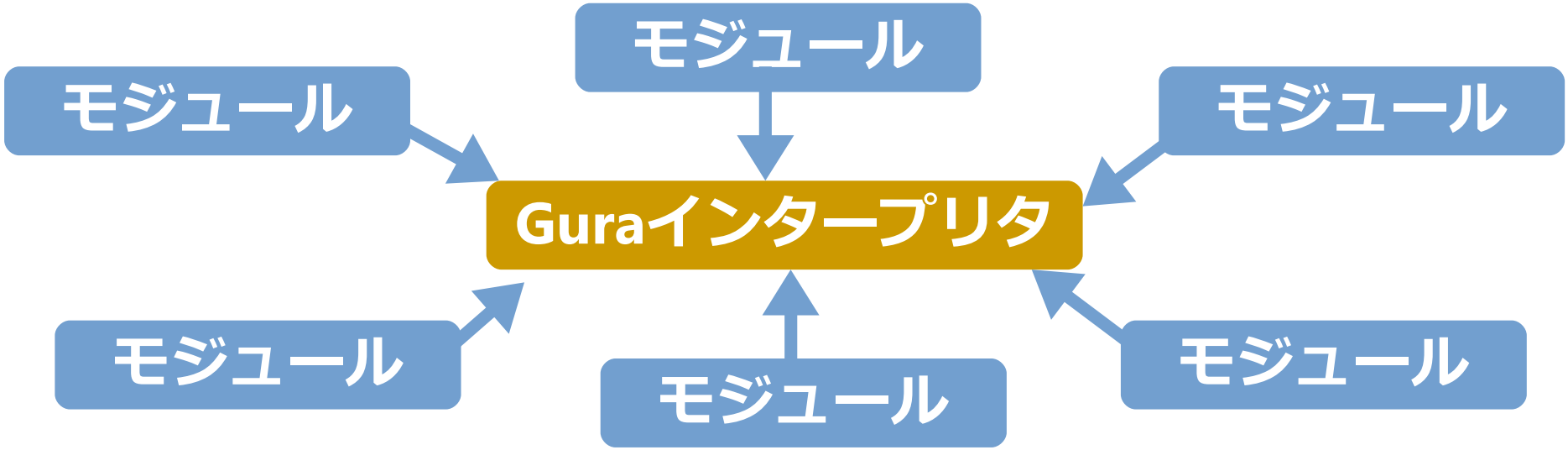
イテレータ処理

拡張モジュール

拡張モジュール

設計ポリシー

- Gura インタープリタ本体は OS 特有の機能やライブラリにできるだけ依存しない
- モジュールを `import` して機能拡張する



おもなモジュール

GUI

wxWidgets　Tk　SDL

グラフィック描画

Cairo　OpenGL　FreeType

イメージデータ

JPEG　PNG　GIF　BMP

ICO　XPM　PPM　TIFF

テキスト処理

CSV　XML　yaml

正規表現　markdown

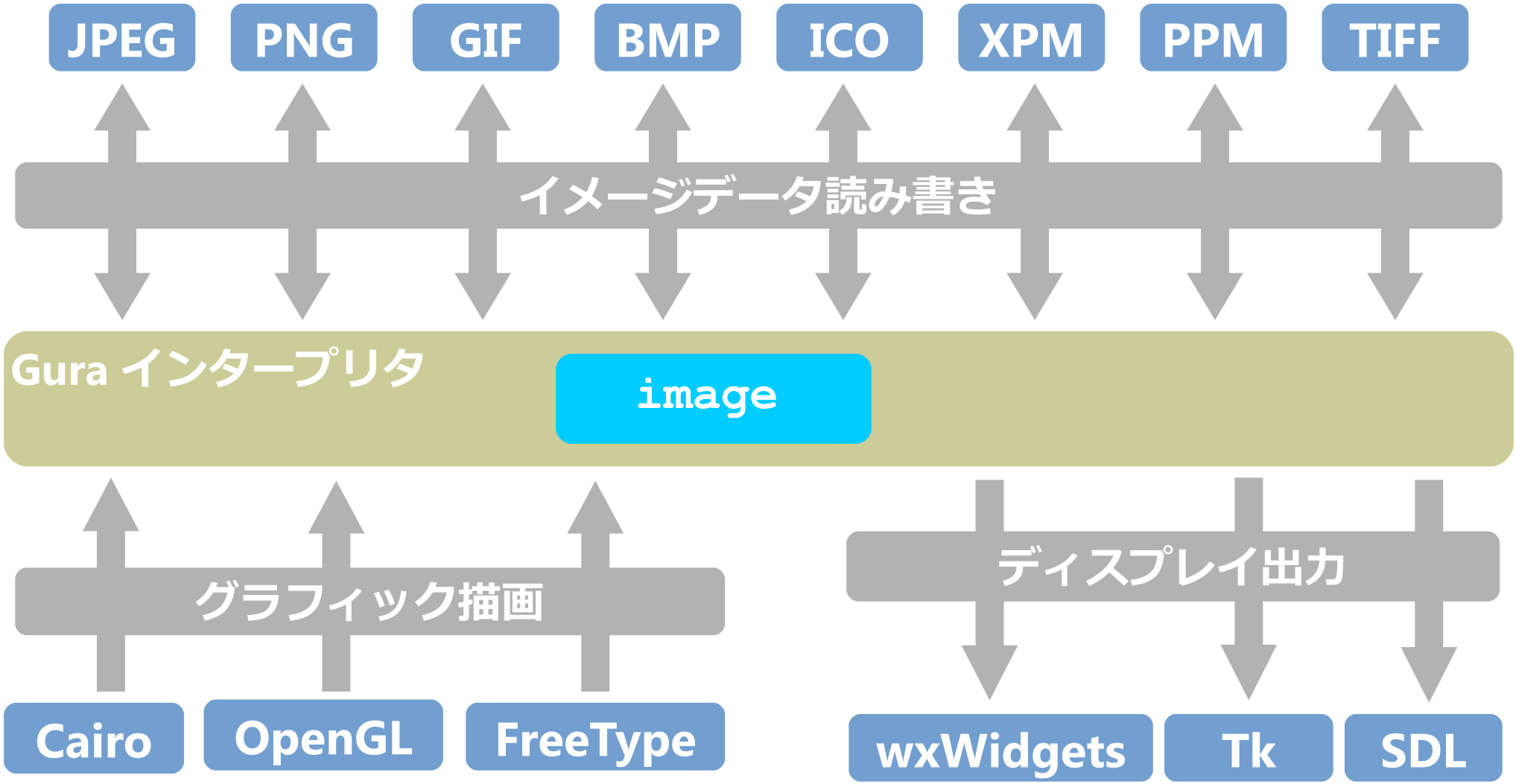
アーカイブ・圧縮

TAR　ZIP　GZIP　BZIP

ネットワーク

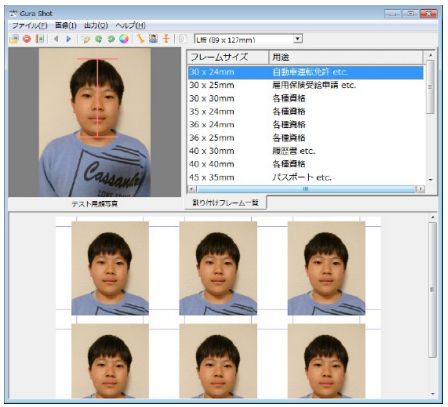
cURL　サーバ機能

モジュール間連携

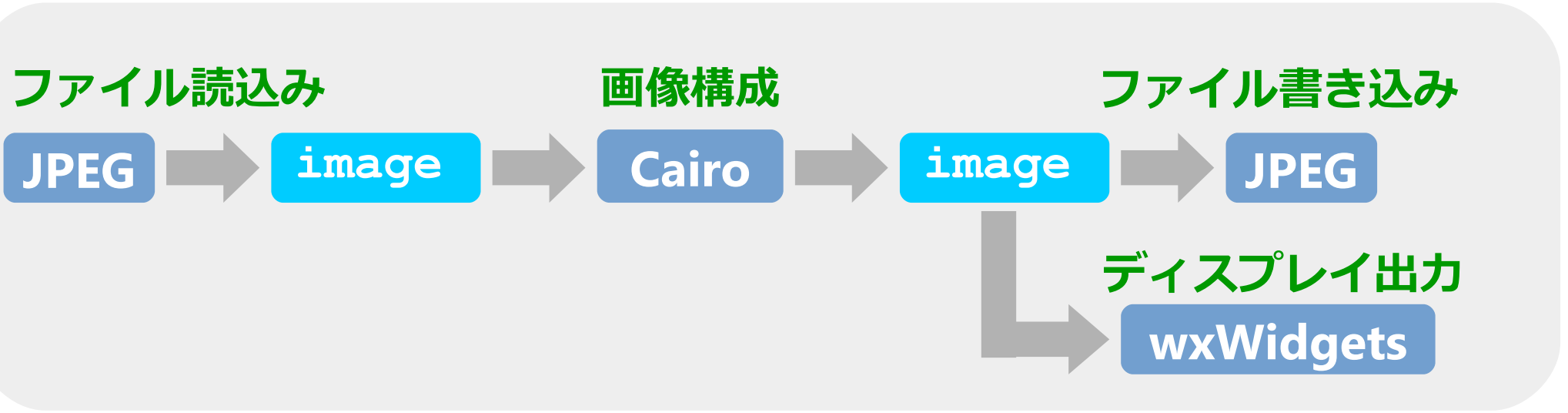


こういうアプリができます

「おうちで証明写真 Gura Shot」



- デジカメ画像から顔を抽出して証明写真を作成
- 結果を PDF や JPEG で出力



ありがとうございました

www.gura-lang.org