

神奈川大学オープンキャンパス2021

模擬講義

「ゲームをつかって学ぶ関数型プログラミング」

理学部 情報科学科 馬谷 誠二

今日つくるゲーム

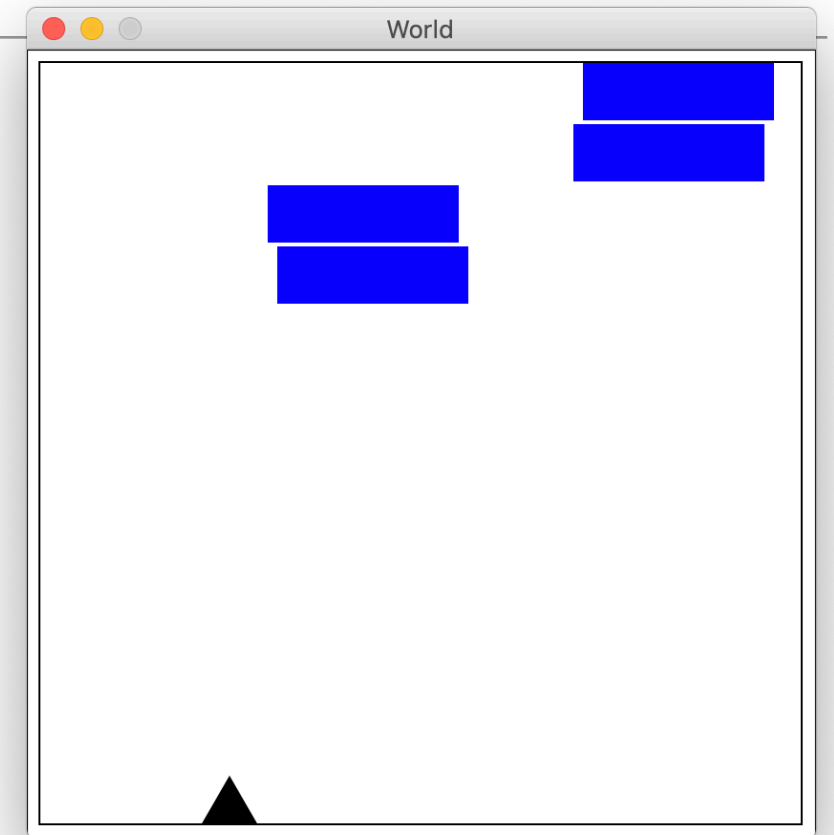
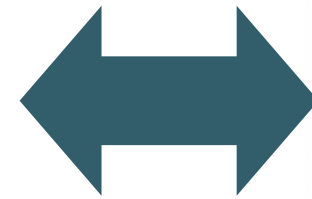
- ...は60分ではムリ！
- 今日はこの程度です(デモ)

授業の主題

最近のほとんどのプログラミング言語は、数学の基本的概念である関数を直接的かつ柔軟に扱う機能を備えています。関数を上手く用いることで、コンピュータの行う複雑な処理の手順（プログラム）を如何に美しく簡潔に書けるかを学びます。この授業では、処理の具体的題材として、みなさんに馴染みのある簡単なゲームを取り上げます。

- 面白いゲームを作る事，ではなく，関数プログラミング入門

有名ゲームと今日つくるゲームの違い

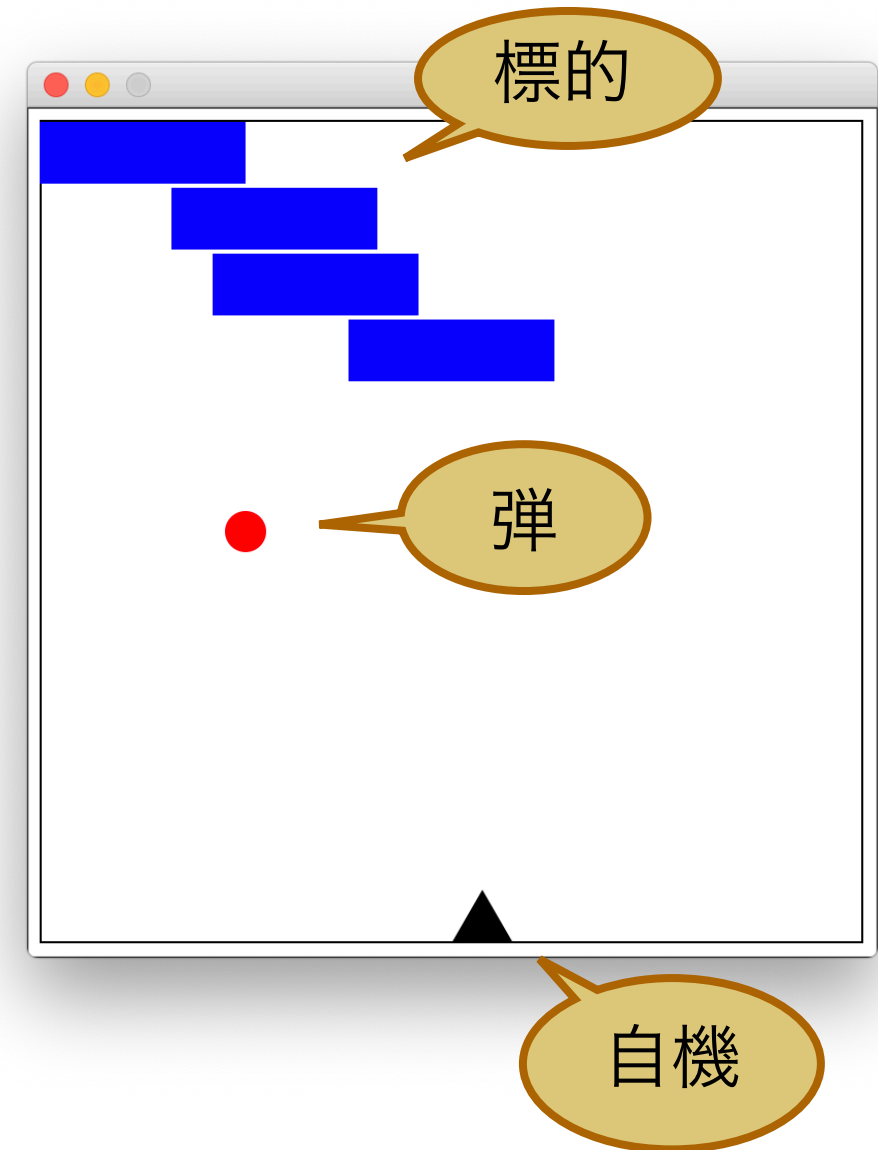


- それでも要素技術は共通（のはず）
 - 図形や画像の描画
 - 時間経過による世界の変化
 - アニメーション
 - キーボード・マウス入力の処理

全然違う！

作成手順

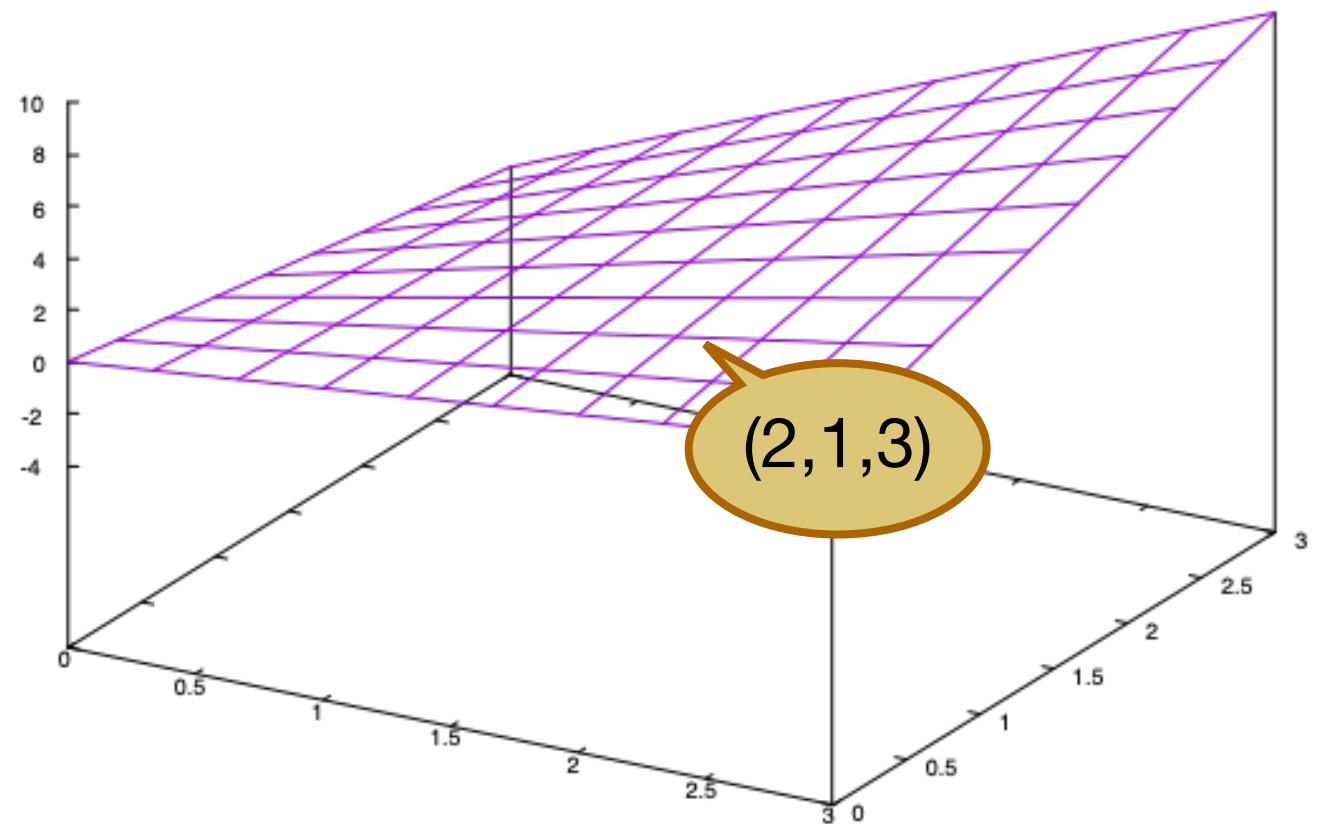
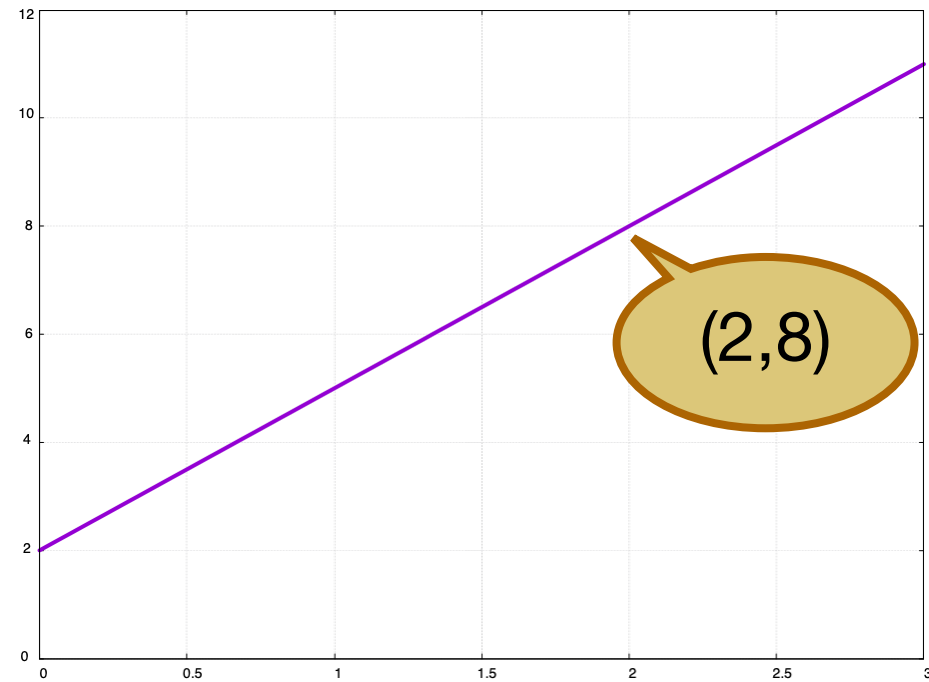
1. 自機の表示と操作
 2. 弾の追加
-
3. 標的の追加
 4. 標的の動作の改良
 5. 弾の連射機能



1. 自機の表示と操作 ～プログラムの基本要素～

高校数学における関数（？）

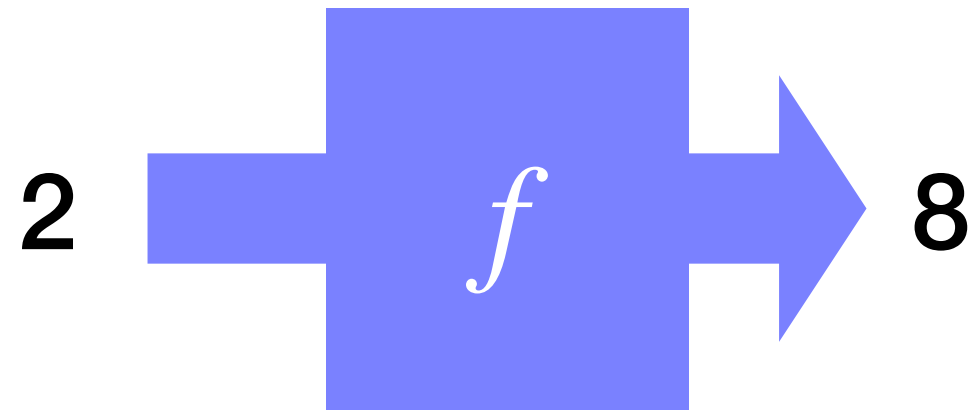
- 1引数関数 $f(x)$
 - $y = f(x) = 3x + 2$
- 2引数関数 $g(x, y)$
 - $z = g(x, y)$
 $= xy + x - y$



計算手続きとしての関数

- 入力を受け取って出力を計算する **処理**（手続き）

$$f(x) = 3x + 2$$



$$g(x, y) = xy + x - y$$



Racket言語

- 関数型プログラミング言語
 - プログラム = データ処理 = 関数による計算
 - 開発環境（インストール&デモ: `basic.rkt`）
 - インタラクション領域
 - 定義領域
 - 扱うデータの種類
 - 数, テキスト, 画像, etc.

数を扱う簡単な関数

- 例: 絶対値関数

- 定義: $|x| = f(x) = \begin{cases} x & (x \geq 0 \text{ のとき}) \\ -x & (\text{それ以外}) \end{cases}$

- 呼出し: $|3| \implies 3, f(-8) \implies 8$

- Racketで呼出しはこう書く (定義は後述)

```
↪ (absolute 3)
3
↪ (absolute -8)
8
↪
```

- 関数には意味のある名前をつける
- 括弧の付け方が全然違う (全体を囲む)

数の四則演算

- 簡単な演算も，実は関数の一種
 - 普段書いている書き方は，関数を呼び出す一般的な書き方の省略形

省略形（数学）

一般形（数学）

Racketコード

$1 + 2$

$+(1,2)$

`(+ 1 2)`

$1 - 2$

$-(1,2)$

`(- 1 2)`

1×2

$\times(1,2)$

`(* 1 2)`

$1 \div 2$

$\div(1,2)$

`(/ 1 2)`

プログラムとは

- 複数の関数呼び出しを組み合わせた式
 - + いくつかの定義（後述）
- 半径3の球の体積： $\frac{4}{3}\pi r^3$, $\div (\times (4, \times (\pi, \times (\times (3, 3), 3))), 3)$
- Racketコード：

```
;; 読みにくい
(/ (* 4 (* 3.141592 (* (* 3 3) 3))) 3)
;; 読みやすい
(/ (* 4 (* 3.141592
              (* 3 3 3)))
   3)
;; 読み間違える
(/ (* 4 (* 3.141592
              (* 3 3 3)))
   3)
```

空白は自由に入れて
かまわない

定数の定義

- 何度も出てくる値に名前をつける → 読みやすくなる
 - 数学でもよく用いる

球の半径 (3) を r , 円周率 (3.141592) を π とする.

球の体積は $\frac{4}{3}\pi r^3$, 表面積は $4\pi r^2$ によって求められる.

- Racketコード (定義エリアのデモ)

```
(define r 3)
(define pi 3.141592)
(define volume (/ (* 4
                    (* pi
                      (* r r r)))
                  3))
(define surface (* 4 (* pi (* r r))))
```

関数の定義

- 特定の球ではなく，任意サイズの球の体積を計算したい！
 - 数学でもよく用いる

円周率 (3.141592) を π とする.

球の体積は 関数 $f(r) = \frac{4}{3}\pi r^3$ によって求められる.

例えば，半径3の球の体積は $f(3) = \frac{4}{3}\pi \cdot 3^3 \doteq 113.1$ である.

- Racketコード (呼び出しをデモ)

```
(define pi 3.141592)

(define (volume r)
  (/ (* 4
        (* pi
            (* r r r)))
     3))
```

$$\pi = 3.141592$$

$$\text{volume}(r) = \frac{4}{3}\pi r^3$$

画像データ

- 色々な幾何図形を生成する関数
 - 例：三角形（デモ）

```
(triangle 50 "solid" "red")
```

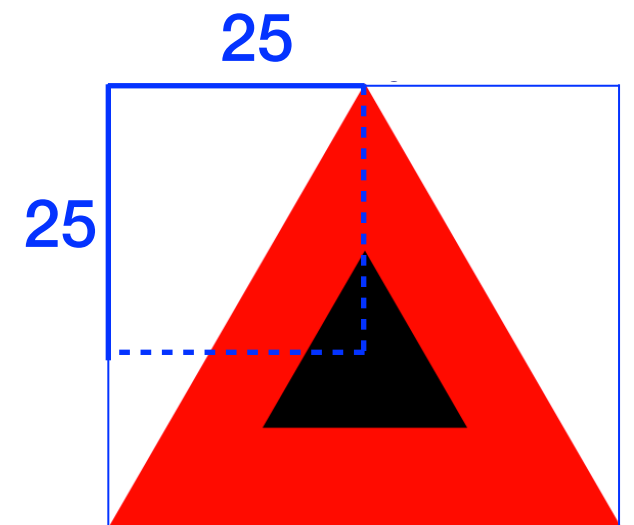
辺の長さ

塗りつぶし

赤

- place-image関数（デモ）
 - 画像に対する計算

```
(place-image (triangle 20 "solid" "black")  
             25 25  
             (triangle 50 "solid" "red"))
```



いよいよゲームの世界へ

- 世界 = 時間とともに**変化していく**状態
 - 状態 = ゲームの世界に存在する**モノの性質**
 - 画面上に見える**全てを記録する必要はない**
- 例1：カウントダウン（デモ）

数

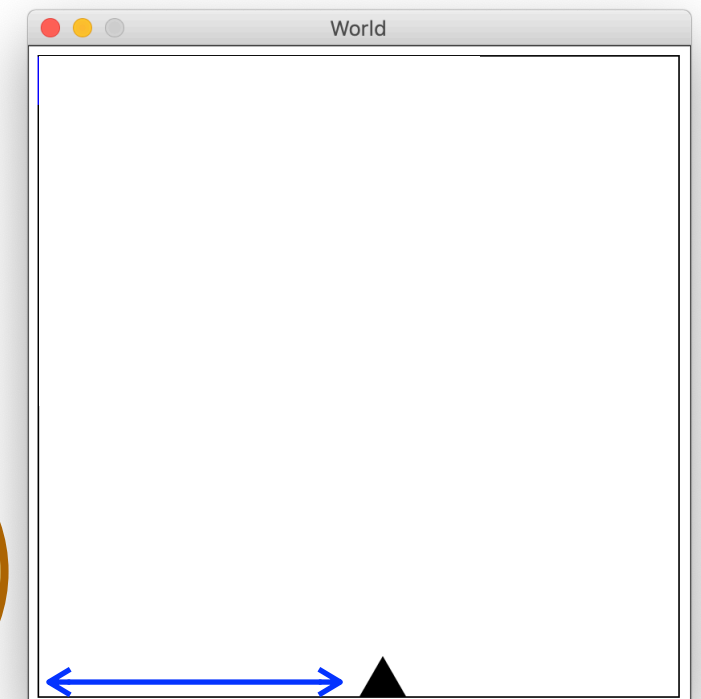
10

- 例2：自機だけからなるシューティングゲーム

位置情報

100

- 自機サイズ
- 三角形であること
- 等々は必要なし



位置情報


ゲーム画面の生成

- 状態（ただのデータ）から画像（見た目）を生成する関数

```
;; ゲーム画面のサイズ
(define SCENE-SIZE 400)
;; ゲーム画面
(define SCENE (empty-scene SCENE-SIZE SCENE-SIZE "white"))
;; 自機
(define ME (triangle 30 "solid" "black"))

;;; 描画処理

(define (draw-scene me-x)
  (place-image ME
    me-x (- SCENE-SIZE
      (/ (image-height ME) 2))
    SCENE))
```



ゲームスタート

- big-bang関数（デモ）
 - 別アプリケーションとして起動
 - 2つの情報を指定
 - 初期状態
 - 状態を描画する関数

```
(define (start)
  (big-bang 100
    [to-draw draw-scene]))
```

キーボード入力によるゲームの操作

- まずは簡単なカウントダウン（デモ）
 - 何らかのキーが押されたら1減らす
- 状態とキーボード情報を受け取り、
変化後の状態を計算する関数

どのキーが押されたかの情報

```
(define (control x k)  
  (- x 1))
```

- big-bang関数で次のように指定

```
(define (start)  
  (big-bang 10  
    [to-draw draw-scene]  
    [on-key control]))
```

自機操作

- 「←」が押された場合と「→」が押された場合では処理が異なる
 - 「←」：自機を5だけ左に動かす
 - 「→」：じきを5だけ右に動かす

cond式

- 場合 (**condition**) 分けして各々で異なる計算
- 数学でもよくある
 - 例：絶対値関数

$$|x| = \begin{cases} x & (x \geq 0 \text{ のとき}) \\ -x & (\text{それ以外}) \end{cases}$$

- Racketコード

```
(define (absolute x)
  (cond [(>= x 0) x]
        [else (- x)]))
```

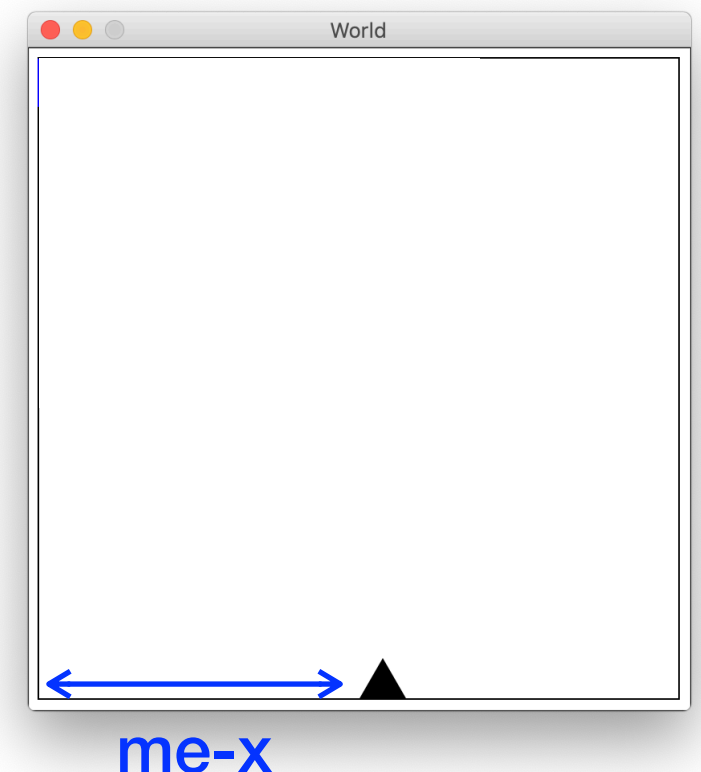
自機操作コード (デモ)

;;;; キーボード入力処理

```
(define (control me-x k)
  (cond [(string=? k "left") (- me-x 5)]
        [(string=? k "right") (+ me-x 5)]
        [else me-x]))
```

;;;; アプリケーションの実行を開始

```
(define (start me-x)
  (big-bang me-x
    [to-draw draw-scene]
    [on-key control]))
```



2. 弾の追加 ～複雑なデータと時間の扱い～

まずはデモ

考えなくてはいけないこと

1. 世界に複数個のモノが存在（自機と弾）

- これまでは1つだけだった
 - カウントダウンの数
 - 自機の位置情報（x座標）
- 状態をどうやって表すか？

2. 何も操作しなくても勝手に変化

→ 時間経過の概念

2つの物体を扱うにはどうする？

- リスト構造
 - 2つのデータを含む1つのデータ
- list関数: 2つのデータを受け取ってリストを作る関数
- first, second関数: リストから各要素を取り出す関数
- Racketコード (デモ)

```
(define x (list 1 "black"))  
  
(first x)  
  
(second x)
```

世界（状態）の定義の更新

- 自機

- 位置情報（x座標）

- 弾

$\left\{ \begin{array}{ll} \text{x,y座標のリスト} & (\text{発射されている場合}) \\ \text{"none"} & (\text{発射されていない場合}) \end{array} \right.$

```
(define (missile x y) (list x y))  
(define (missile-x m) (first m))  
(define (missile-y m) (second m))
```

- 世界

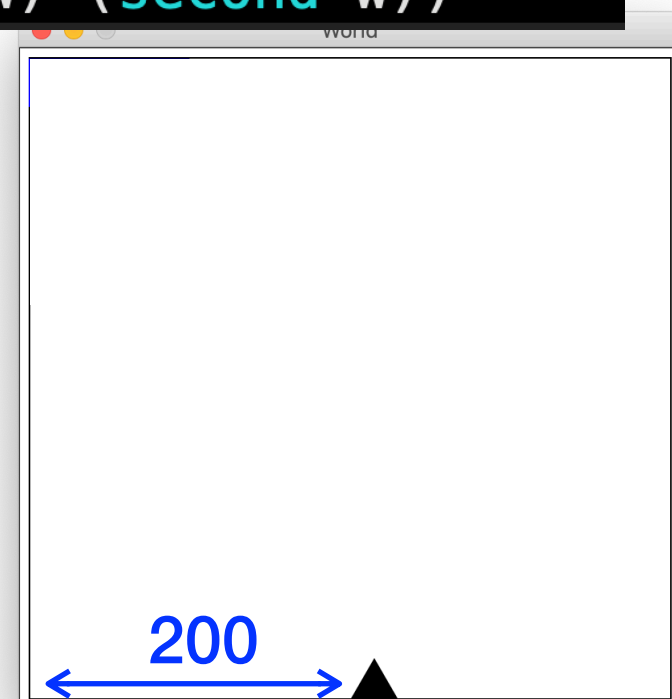
- 自機と弾のリスト

```
(define (world me-x mi) (list me-x mi))  
(define (get-me-x w) (first w))  
(define (get-missile w) (second w))
```

使用例（デモ）

```
(define m (missile 100 250))  
(define w (world 200 m))  
(define w2 (world 200 "none"))
```

w2



発射キー入力処理の追加（デモ）

;;; キーボード入力処理

```
(define (control w k)
  (cond [(string=? k "left")
        (world (- (get-me-x w) 5)
                (get-missile w))]
        [(string=? k "right")
        (world (+ (get-me-x w) 5)
                (get-missile w))]
        [(string=? k " ")
        (world (get-me-x w)
                (cond [(string? (get-missile w))
                      (missile (get-me-x w)
                               (- SCENE-SIZE (image-height ME)))]
                      [else (get-missile w)]))]
        [else w]))
```

ミサイル情報はそのまま
自機のx座標を-5

自機のx座標はそのまま
ミサイルを自機のすぐ上に発射
(発射済みならそのまま)

;;; アプリケーションの実行を開始

```
(define (start me-x)
  (big-bang (world me-x "none")
            [to-draw draw-scene]
            [on-key control]))
```

時間経過処理

- 一定時間（1/28秒）が経過する毎に任意の関数を呼び出すことが可能
 - 状態を少しだけ変えると動いてるように見える

```
(define (start me-x)
  (big-bang (world me-x "none")
    [to-draw draw-scene]
    [on-tick next]
    [on-key control]))
```

- next関数：状態を受け取り，変化後の状態を計算
 - 例：カウントダウン（デモ...しても見えない）

```
(define (next x)
  (- x 1))
```

弾を動かすためのコード

ミサイルのy座標を-8する関数

```
;;;; 世界の状態を更新する処理  
(define (move-missile m)  
  (missile (missile-x m) (- (missile-y m) 8)))
```

弾を動かすためのコード

```
;;;; 世界の状態を更新する処理
(define (move-missile m)
  (missile (missile-x m) (- (missile-y m) 8)))

(define (next w)
  (world (get-me-x w)
    (cond [(string? (get-missile w)) "none"]
          [(>= (missile-y (get-missile w)) 0)
           (move-missile (get-missile w))]
          [else "none"])))
```

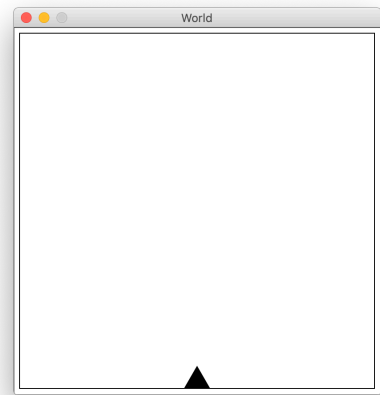
ミサイルが発射されていない場合

ミサイルが画面の上端まで到達していない場合

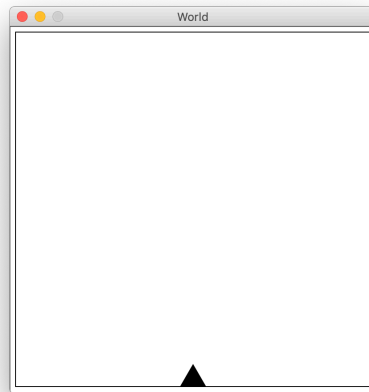
ミサイルが画面の上端に到達した場合

まとめ：状態変化の時系列

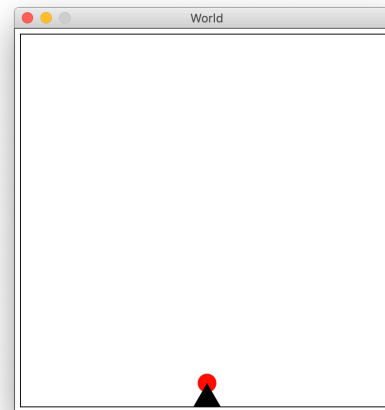
初期状態



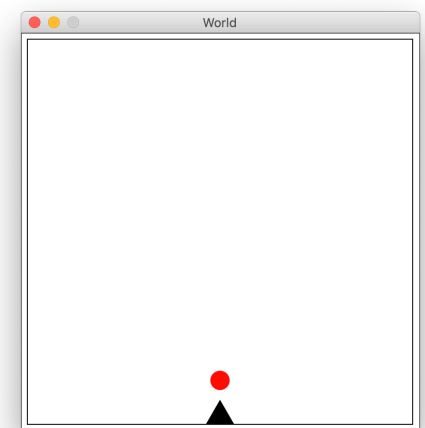
next関数



control関数



next関数



あらゆる状態変化を
関数で表現

時刻 t

今日はここまで

「プログラミングB/B演習」

- 情報科学科 2年前期科目
- リアクティブ・プログラミング
 - 自由課題：ゲーム制作
 - 受講生の作成したゲームの例

最後に一言：関数を究める

- コンピュータ上で行われる全ての処理は関数で記述可能
 - 伝統的なスタイルに比べ，シンプルで理解しやすい

- λ -calculus

- 処理だけでなく，データも含め全てが関数
 - 例：自然数

```
(define (zero f z) z)
(define (one f z) (f z))
(define (two f z) (f (f z)))

(define (plus m n)
  (define (k f z) (m f (n f z)))
  k)
```

- 大学の学問
 - 一見役に立ちそうに見えなくても「計算すること」の本質を追求

参考情報

- Racketホームページ（英語）
 - <https://racket-lang.org>
 - ソフトウェアのダウンロード, マニュアル
- このスライド・ソースコード
 - <https://github.com/umatani/ku-oc.git>
- Schemeの簡単な入門書（日本語訳）
 - <https://www.sampou.org/scheme/t-y-scheme/t-y-scheme.html>
- Schemeを使った情報科学の入門書（世界的に有名）
 - <https://github.com/hiroshi-manabe/sicp-pdf/blob/japanese/jsicp.pdf>
- 私のメールアドレス
 - umatani@kanagawa-u.ac.jp