

実践情報セキュリティと

アルゴリズム 2

プログラミング言語Rust

大阪大学 大学院工学研究科 電気電子情報通信工学

王 イントウ

wang@comm.eng.osaka-u.ac.jp

Q&A

バッファオーバーランについて

- Q：バッファオーバーランは、CTFで演習するバッファオーバーフローとどう違うのか？アドレス空間配置のランダム化して連続領域でない場合も起こるのでしょうか？
- A：バッファオーバーラン = バッファオーバーフローです。また、アドレス空間配置ランダム化機能はプロセスが利用するメモリ空間を連続領域ではなくランダムに分散させて配置する方法で、バッファオーバーランなどのセキュリティ脆弱性が発生した場合のセキュリティとして機能する。

書籍

本講義で用いる書籍

難しすぎるので読む必要なし

- Benjamin C. Pierce、『Advanced Topics in Types and Programming Languages』、The MIT Press

- 高野 祐輝、『並行プログラミング入門』、オライリー・ジャパン
正誤表：https://github.com/oreilly-japan/conc_ytakano/blob/main/errata.md
- Hillel Wayne、『実践TLA+』、翔泳社

おすすめです

Introduction

Rust言語

- 元々は、Firefoxなどを作っているMozillaで働いていた、グレイドン・ホアレの個人プロジェクト
- 今はMozillaが積極的に開発しており、Firefoxなどで利用されている
- 特徴
 - 利点：安全、速い、GCがない
 - 欠点：まだそれほど広まっていない、型システムのパラダイムが新しく、とっかかりにくい

Rustの利用例

- Rust Production Users
<https://www.rust-lang.org/production/users>
- Google Chrome OS
https://chromium.googlesource.com/chromiumos/docs/+/master/rust_on_cros.md
- LINE：設定エージェントで利用
- DWANGO：いくつかのRustライブラリを実装し，分散オブジェクトストレージを構成

なぜRust言語を学ぶか？

- ・ 疑問：なぜRust言語なのか。まだそれほど広まっていないし、他のIT技術のように数年で置き換わるのでは？
- ・ **×**Rust言語を学ぶ
- ・ **○**Rust言語を通して、安全なソフトウェアを実現する現代的な機能と、その理論的背景を学ぶ
 - ・ たとえ、Rustが廃れても、その考えは後続技術に受け継がれる
 - ・ 重要なのは理論と哲学を学ぶ事

Rustで書いても他に脆弱性があるのでは

- 疑問：結局Rustで書いても、他のCなどで書かれた場所があると、そこが問題になるのでは？
- 回答：そのとおり
都市計画と同じで、すぐにすべてを置き換えるのは無理
10年、20年の長期的な視点が必要で、それを行うのが、コンピュータサイエンスの素養があり、未来のIT技術を担う本講義の受講生
- 将来どうしたいか、どうあるべきかというビジョンを持ち実現して行こう

型システム

プリミティブ型 (1/2)

ビット長	符号付き整数	符号なし整数
8	i8	u8
16	i16	u16
32	i32	u32
64	i64	u64
128	i128	u128
環境依存	isize	usize

プリミティブ型 (2/2)

型	説明	リテラル
f32	32ビット浮動小数点数	3.14
f64	64ビット浮動小数点数	3.14
bool	真偽値	true, false
char	文字	'a', 'あ'
(型, 型, ...)	タプル	関数の引数タプル (bool, i32) or (true, 10)
[型; 整数値]	配列	xs [i32; 10] = [3; 10] は3が10個の配列 or ys [i32; 3] = [3, 5, 7] は3要素の配列

enum型

```
enum Memory {  
    Mem8G,  
    Mem16G,  
    Mem32G,  
}
```

Memoryという、新しいenum型を定義
8GiB、16GiB、32GiBの**いずれか**

enum型の名前とラベルは大文字から始めるのが通例（キャメル記法）

値を持つenum型

```
enum Dimension {  
    Dim2(u32, u32),  
    Dim3(u32, u32, u32),  
}
```

Dim2の場合、2つの値を持つ

Dim3の場合、3つの値を持つ

struct型

```
struct Computer {  
    memory: Memory,  
    manufacture: String,  
}
```

Computerという、新しいstruct型を定義

Memory型のmemoryというメンバ変数と、
String型のmanufactureというメンバ変数をもつ

struct型の型名は、大文字からはじめ（キャメル記法）、
メンバ変数は、小文字からはじめるのが（スネーク記法）通例

命名規則

- ・ Rustでは以下の2種類が分けて利用される。
- ・ キャメル記法
 - ・ 単語のはじめを大文字にして書く記法。ラクダっぽく見えるのでキャメル
 - ・ 例：RedApple、LockVarなど
- ・ スネーク記法
 - ・ 単語の間をアンダースコア_で区切る記法。蛇っぽいのでスネーク
 - ・ 例：red_apple、lock_var
- ・ 命名規則に沿わない場合Warningがでる

Parameterized型（ジェネリクス）

```
struct Point<T> {
```

Tが型引数で、Tに様々な型を代入可能

```
    x: T,  
    y: T,  
}
```

例

```
Point<u32>
```

Tがu32に置き換わった、Point型となる
つまり、上のxとyの型がu32となる

enumにも同じように適用可能

参照型

型名の前に、&をつけるとimmutable（破壊的代入ができない）な参照型で、&mutをつけるとmutable（破壊的代入可能）な参照型

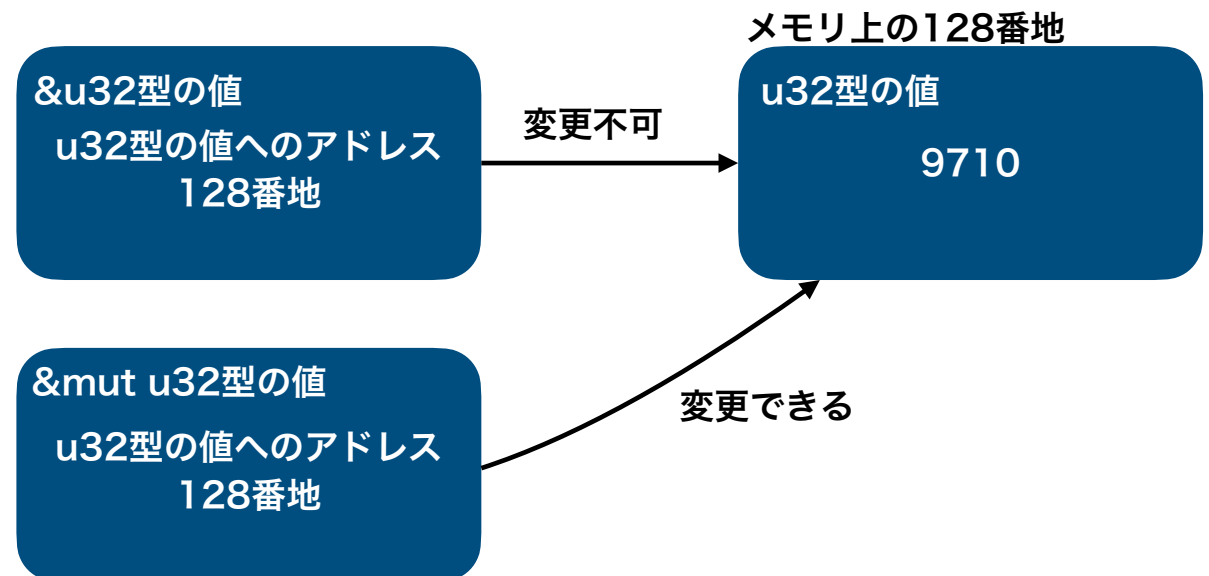
例

`&u32`

`&mut u32`

`&Point<f64>`

`&mut Gender`



Option型

Option型の定義

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

Option型はRustやOCaml、（HaskellだとMaybe型）など、安全な言語で頻繁に用いられる型なのでマスターしておく事。

利用例：nの一個前の数を返す関数

```
fn predecessor(n: u32) -> Option<u32> {  
    if n == 0 {  
        None  
    } else {  
        Some(n - 1)  
    }  
}
```

Noneか、Someに包まれたn - 1を返す。

Result型

Result型の定義

```
enum Result<T1, T2> {  
    Ok(T1),  
    Err(T2),  
}
```

Rustでよく使う型。失敗した場合の原因を返す事ができる。

```
fn predecessor(n: u32) -> Result<u32, String> {  
    if n == 0 {  
        Err("n is zero".to_string())  
    } else {  
        Ok(n - 1)  
    }  
}
```

Err(失敗した原因の文字列)か、Ok(n - 1)を返す。

パターンマッチ

Result型（かMaybe型）は、必ず値を検査して中身を取り出す必要がある。

エラーハンドリングをしない記述は（ほぼ）あり得ない。

```
fn pat() {  
    match predecessor(0) {  
        Ok(n) => {  
            println!("success: {}", n);  
        }  
        Err(s) => {  
            println!("error: {}", s);  
        }  
    }  
}
```

成功の場合

失敗の場合

unwrap

エラーハンドリングを少しサボる

unwrapでSomeかOk中の値を検査して中身を取り出す。

エラーの場合プログラムは停止

```
fn uw() {  
    let n = pred(0).unwrap();  
    println!("success: {}", n);  
}
```

エラーハンドリングの哲学

- Rustが許容するエラーハンドリング方法
 - エラーハンドリングを正しく行う
 - もしくは、エラーハンドリングをちょっとサボるが、失敗した場合はプログラムが停止する
 - OCamlやHaskellなどもこちら
- Rustが受け入れないエラーハンドリング方法
 - エラーハンドリングをせずに、失敗したことに気がつかず、成功した時と同じ処理をそのまま進めてしまう
 - Python、Ruby、C、C++、Java、JavaScriptなどの言語ではこちら

良いエラーハンドリング

成功  成功した場合の処理

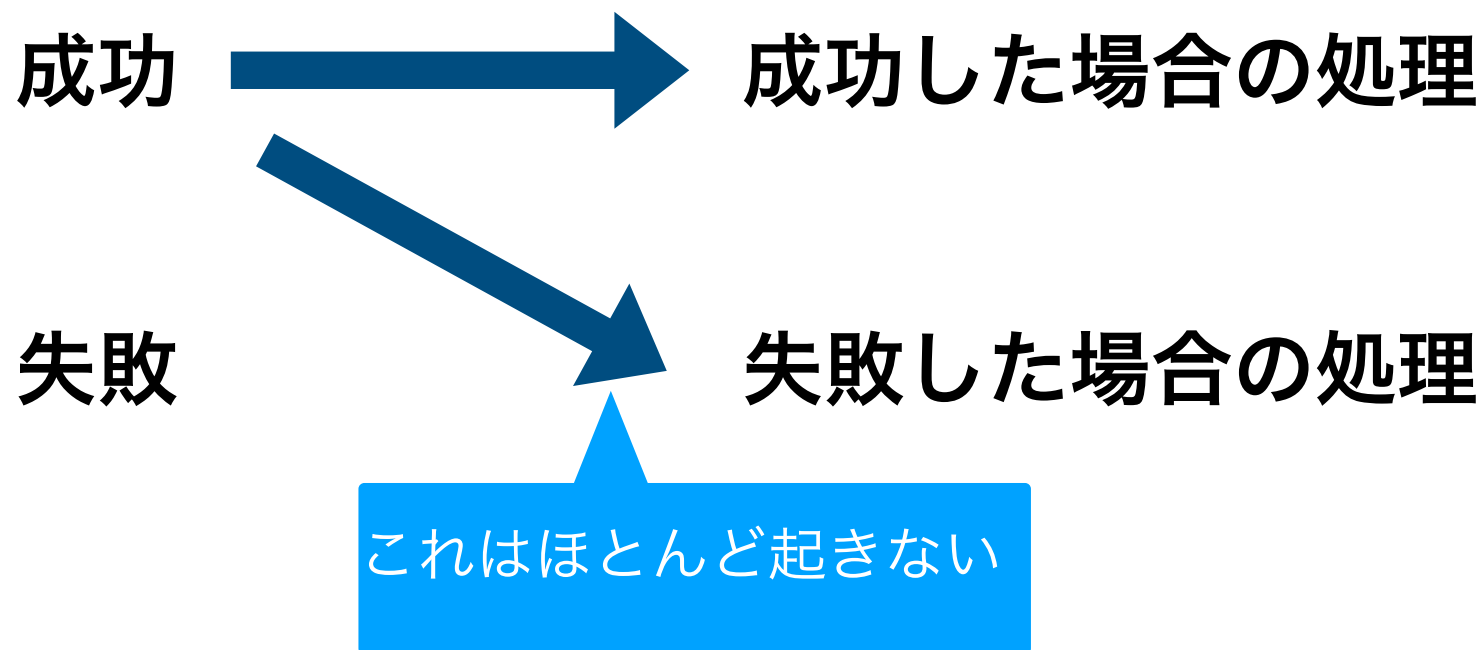
失敗  失敗した場合の処理

ダメなパターン



これを防ぎたい。Rustなどの場合、これは型検査で防ぐ事が可能。

ほとんど起きないパターン



基本的な文法

let文

let文でimmutable変数定義となり、let mutでmutable変数定義となる

```
let x = 100;  
let mut y = 20; // mutable変数  
let z: u32 = 5; // 明示的に型を指定可能  
y *= x + z; // y は mut で宣言されているため、破壊的代入可能  
let w;  
w = 8;
```

関数定義

関数定義は、

fn 関数名 (引数名 : 型、...) -> 戻り値の型 { 関数の中身 }

と書く

!のつく関数は、関数ではなくてマクロ。

println!マクロは、第一引数の{}に第二引数以降の値を表示

```
fn hello(v: u32) {  
    println!("Hello World!: v = {}", v);  
}
```

関数の戻り値の型は -> の後ろに書く

```
fn add(x: u32, y: u32) -> u32 {  
    x + y  
}
```

セミコロンのない式は、値を返す式となる
(PythonやC言語のreturnっぽい感じ)

```
fn my_func1() {  
    let n = add(10, 5);  
    hello(n);  
}
```

if式

if式はCやJavaScriptなどと似ているが、条件の式に()は必要なく、式なので値を返す

ただし、以下の型的な制約が求められる：

- ・ 条件の式はbool型であること
- ・ ifの返り値は全て同じ型であること

```
fn is_even(v: u32) -> bool {  
    if v % 2 == 0 {  
        true  
    } else {  
        false  
    }  
}
```

条件式はbool型でなければならない

trueとfalseは両方とも同じbool型なのでOK

struct型のオブジェクト生成

struct型は、

型名{変数名: 値, ...}

として定義。メンバ変数は必ず初期化が必要。

```
struct Point {  
    x: u64,  
    y: u64,  
}  
  
fn st() {  
    let p = Point{x: 100, y: 200};  
}
```


enum型のオブジェクト生成

enum型は、

型名::ラベル(値、…)

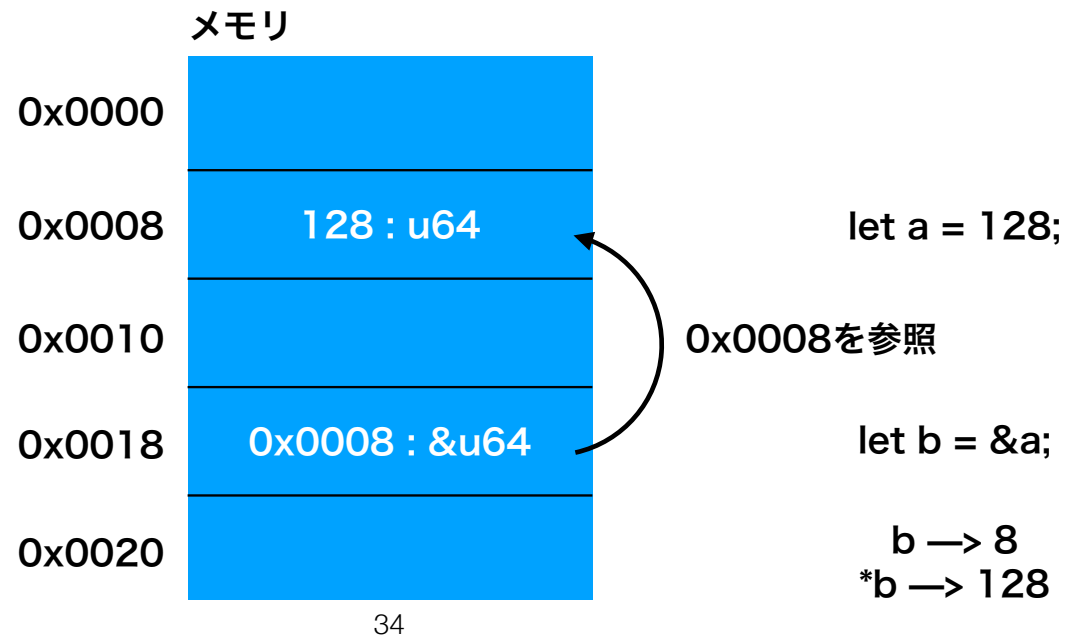
として定義。値を持たない場合は、カッコを省略可能。

```
enum Dimension {  
    Dim2(u32, u32),  
    Dim3(u32, u32, u32),  
}
```

```
fn dm() {  
    let d = Dimension::Dim3(40, 50, 7);  
}
```

参照

- C言語のポインタと同じと思えば良い
- 実体ではなく、アドレスを保持している



参照

- ・ 変数に&をつけるとimmutable参照を取得し、&型名でimmutableな参照型
- ・ &の代わりに&mutとすると、mutable参照となる
- ・ *参照変数で、参照外し

```
fn mul(x: &mut u64, y: &u64) {  
    // (*x) = (*x) * ((*x) * (*y)) という意味  
    *x *= *x * *y;  
}  
  
fn my_func2() {  
    let mut n = 10;  
    let m = 20;  
    println!("n = {}, m = {}", n, m); // n = 10, m = 20  
    mul(&mut n, &m);  
    println!("n = {}, m = {}", n, m); // n = 2000, m = 20  
}
```

match式

条件を書いて分岐が可能

```
fn pred(v: u32) -> Option<u32> {  
    if v == 0 {  
        None  
    } else {  
        Some(v - 1)  
    }  
}
```

```
fn print_pred(v: u32) {  
    match pred(v) {  
        Some(w) => {  
            println!("pred({}) = {}", v, w);  
        }  
        None => {  
            println!("pred({}) is undefined", v);  
        }  
    }  
}
```

matchの|とワイルドカード

|で複数条件(or)を記述でき、_を使うと(その他)全ての条件にマッチする

```
enum ABC {  
    A,  
    B,  
    C,  
    D,  
    E,  
}
```

```
fn mc(abc: ABC) -> u64 {  
    match abc {  
        ABC::A => 1,  
        ABC::B | ABC::C => 2,  
        _ => 0,  
    }  
}
```

for文

forは、繰り返しできる要素に対して、順に処理を行う。

Cよりも、pythonのforに近い

配列などの繰り返し可能なオブジェクトは、iter()か、&で取得

```
fn fr() {  
    let v = [1, 4, 5];  
    for i in v.iter() { // v.iter() の代わりに &v としてもOK  
        println!("i = {}", i);  
    }  
}
```

for文 破壊的代入

破壊的代入をしつつ、繰り返しを行うには、`iter_mut()`か`&mut`を利用

```
fn fr_mut() {  
    let mut v = [1, 4, 5];  
    for i in v.iter_mut() { // v.iter_mut() の代わりに &mut v としてもOK  
        *i = *i + 1;  
        println!("i = {}", i);  
    }  
}
```

loop文

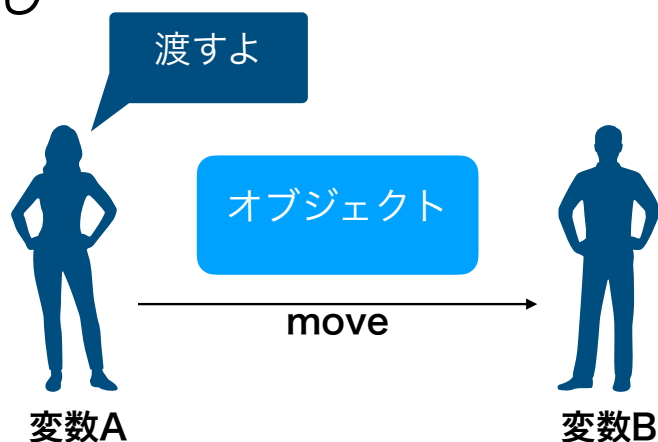
無限ループ。break抜けるか、returnで関数ごと抜けないと終了しない

```
fn app_n(f: fn(u64) -> u64, mut n: u64, mut x: u64) -> u64 {  
    loop {  
        if n == 0 {  
            return x;  
        }  
        x = f(x);  
        n -= 1;  
    }  
}
```


所有権・ライフタイム・借用

所有権

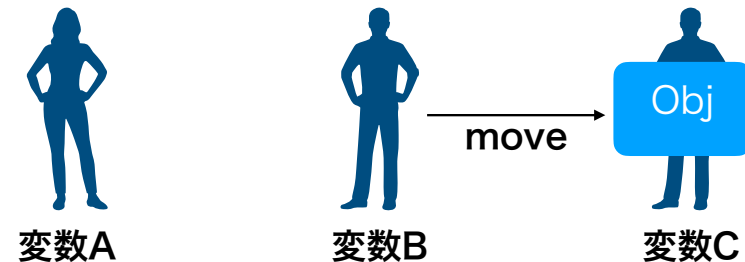
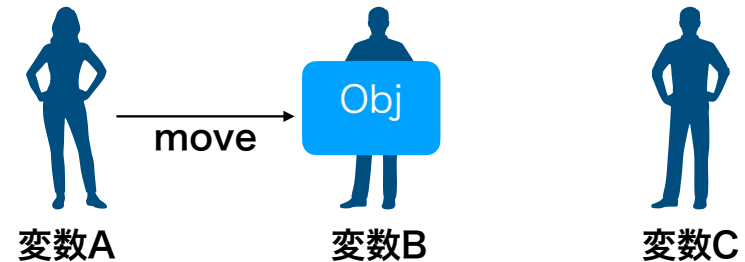
- Rustはデフォルトでは、変数から変数にオブジェクトが移動するような動作をする
- C++のスマートポインタ (unique_ptr) にも、同じような考えはある
- バケツリレーっぽい感じ



Move

```
#[derive(Debug)]  
struct Obj{}
```

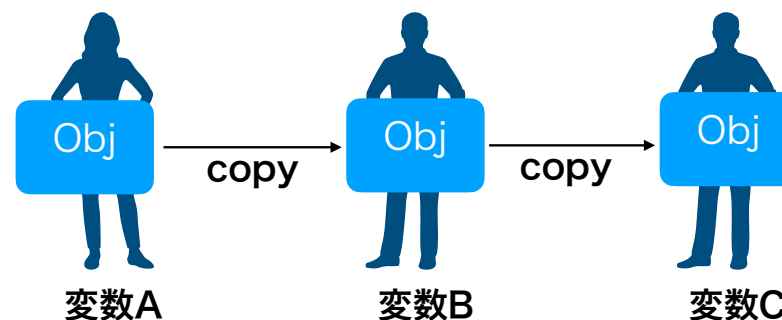
```
fn mv() {  
    let a = Obj{};  
    let b = a;  
    let c = b;  
    // let d = b; これはエラー  
    println!("{:?}", c);  
}
```



Moveの対象とならない型

- Copy trait (traitは後ほど説明) が実装されている型は、moveではなくて、copyが基本になる
- boolやu32などのプリミティブ型はcopy

```
fn main() {  
    let a = 10;           // immutable object  
    let b = a;            // copy  
    let c = b;            // copy  
    println!("{}", a, b, c); // borrow check! - OK  
}
```

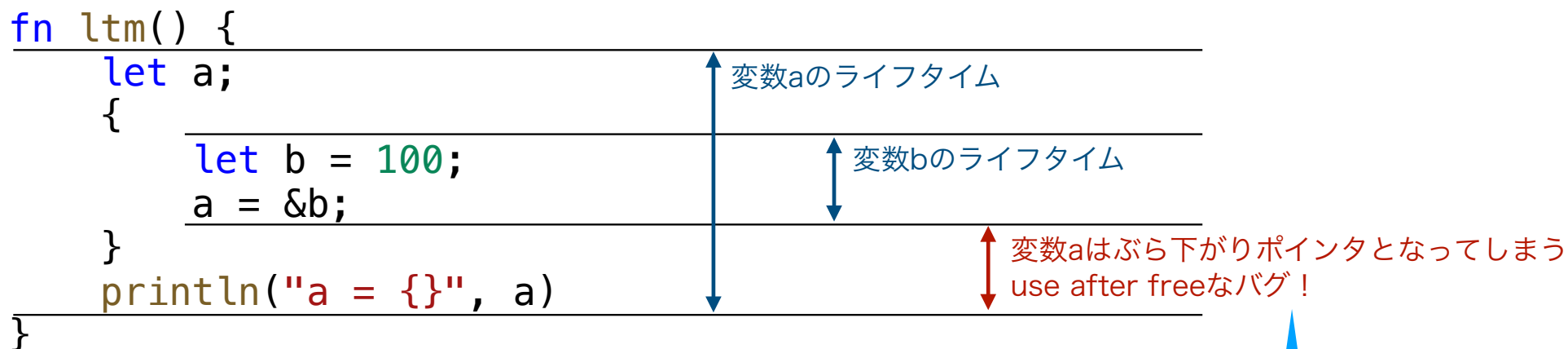


ライフタイム

- 人に寿命があるように、変数にも寿命がある
- 変数の寿命とは？
 - let、match、引数で定義されてから、スコープが外れる（もしくは最後に使われる）まで
 - Rustの全ての変数には、暗黙的にライフタイムが設定されている

ライフタイムの例

コンパイルエラーとなる例



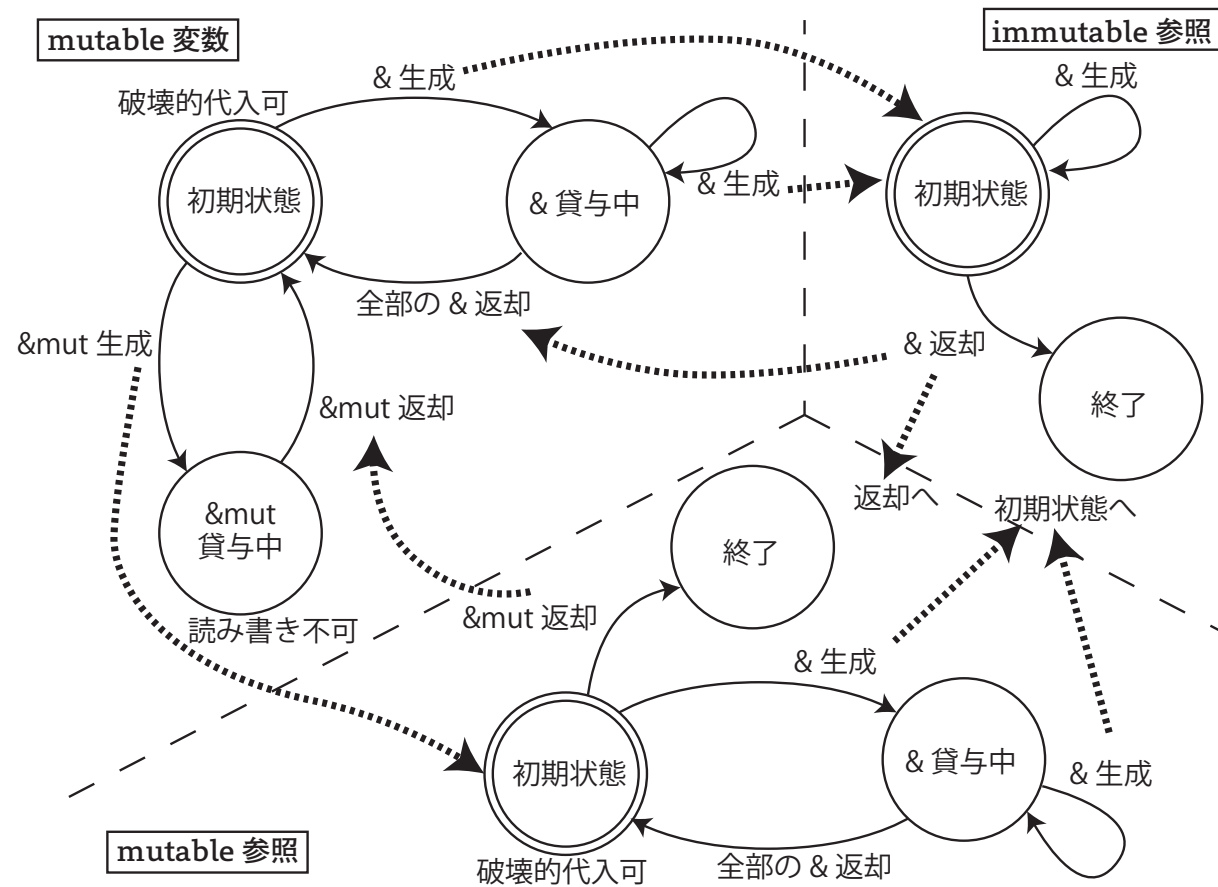
変数aのライフタイムは、変数bより長いため、
変数bの参照を変数aに代入する事はできない。

これを防ぎたい。Rustの場合、型検査で防ぐ事が可能。

借用

- ・ 以下の2つを保証するためのメカニズム
 - ・ あるオブジェクトに、破壊的代入を行える変数は、同時に2つ以上存在しない
 - ・ ある時刻で、あるオブジェクトに対して破壊的代入を行える変数がある場合、その時刻ではその変数以外に、読み書き可能な変数は存在しない
- ・ なぜこんなことをやるのか？
 - ・ マルチスレッドプログラミングを容易にするため
 - ・ つまり、更新可能な共有情報を極力減らす
 - ・ 共有メモリがバグの元

借用時の変数の状態（簡易モデル）



(完璧ではないが、学習には良いモデル)

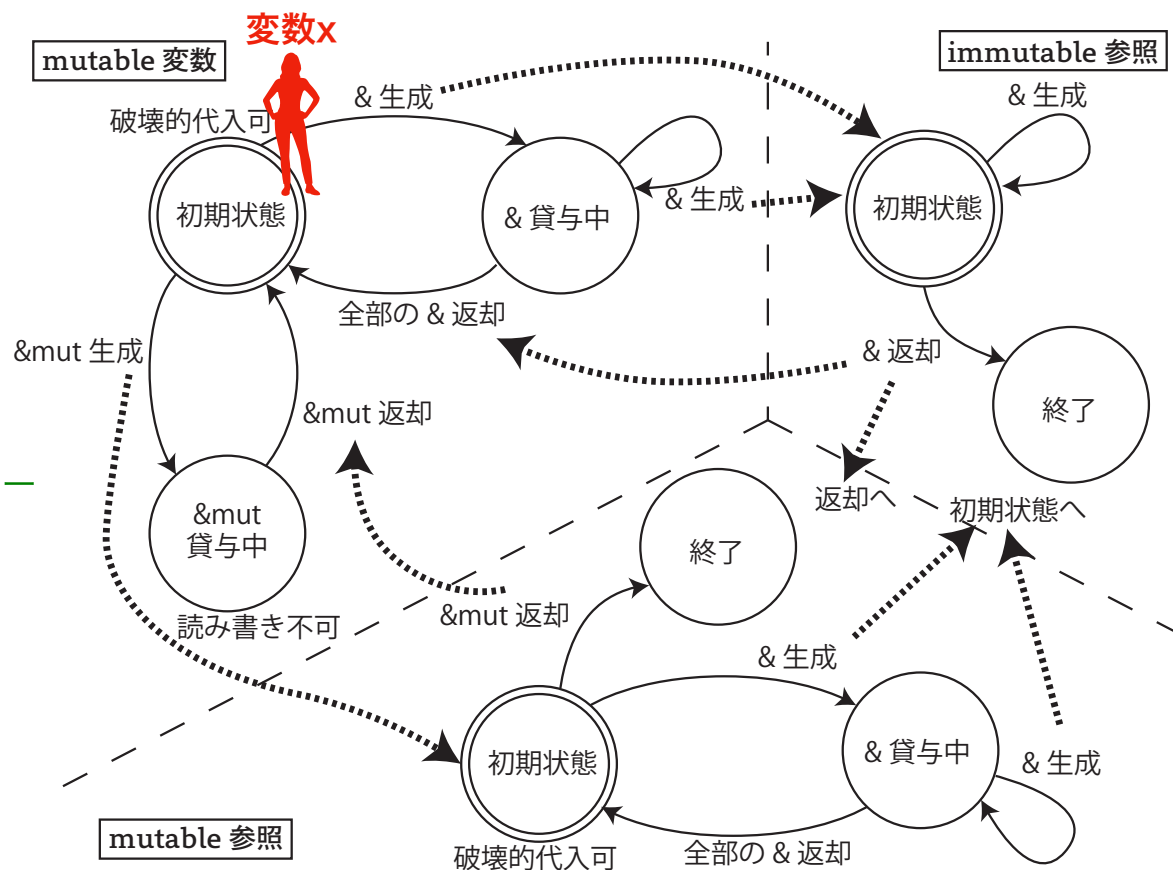
借用の例 (1/5)

```
let mut x = Foo{val: 10}; // xはmutable変数
{
  let a = &mut x; // aはmutable参照
  println!("a.val = {}", a.val);

  // xは「&mut貸与中」状態のためエラー
  // println!("x.val = {}", x.val);

  let b: &Foo = a; // bはimmutable参照
  // a.val = 20;    // aは「&貸与中」状態のためエラー
  println!("b.val = {}", b.val);
  // ここでbが返却される

  a.val = 30;
}
```



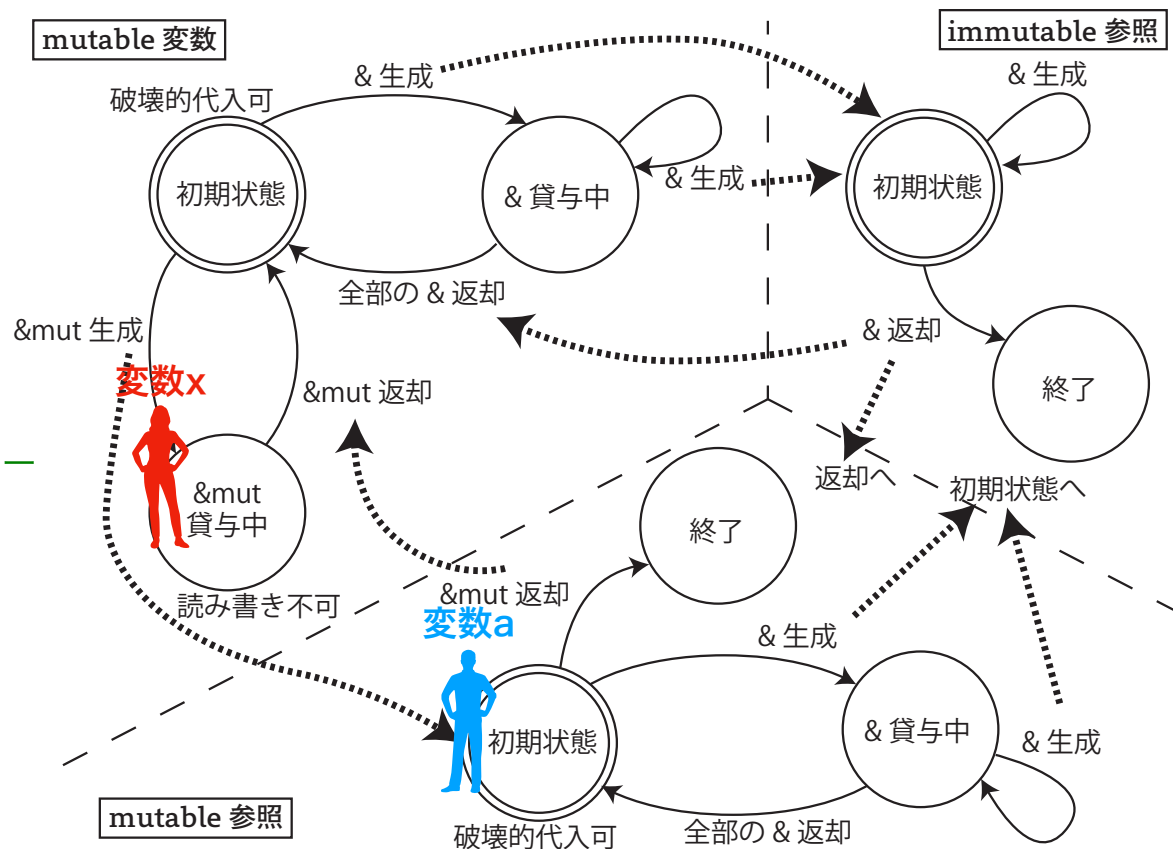
借用の例 (2/5)

```
let mut x = Foo{val: 10}; // xはmutable変数
{
  let a = &mut x; // aはmutable参照
  println!("a.val = {}", a.val);

  // xは「&mut貸与中」状態のためエラー
  // println!("x.val = {}", x.val);

  let b: &Foo = a; // bはimmutable参照
  // a.val = 20;    // aは「&貸与中」状態のためエラー
  println!("b.val = {}", b.val);
  // ここでbが返却される

  a.val = 30;
}
```



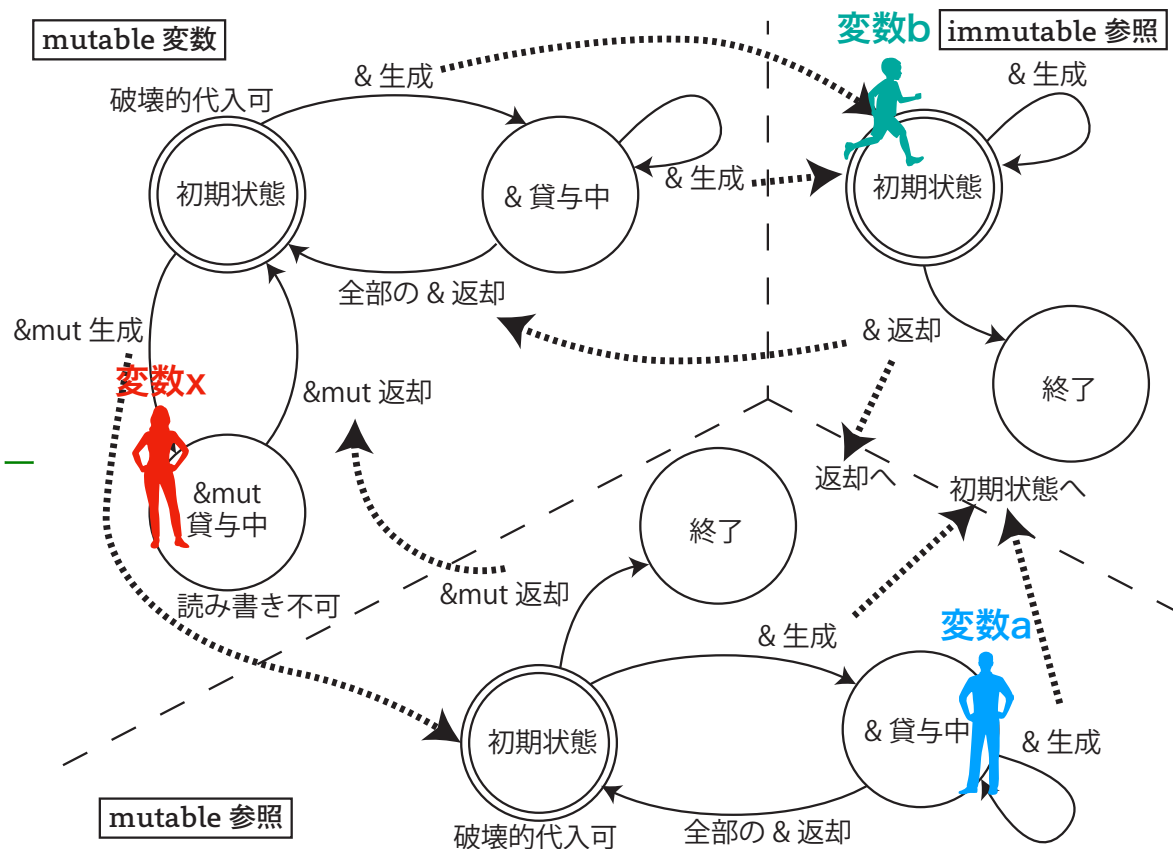
借用の例 (3/5)

```
let mut x = Foo{val: 10}; // xはmutable変数
{
  let a = &mut x; // aはmutable参照
  println!("a.val = {}", a.val);

  // xは「&mut貸与中」状態のためエラー
  // println!("x.val = {}", x.val);

  let b: &Foo = a; // bはimmutable参照
  // a.val = 20;    // aは「&貸与中」状態のためエラー
  println!("b.val = {}", b.val);
  // ここでbが返却される

  a.val = 30;
}
```



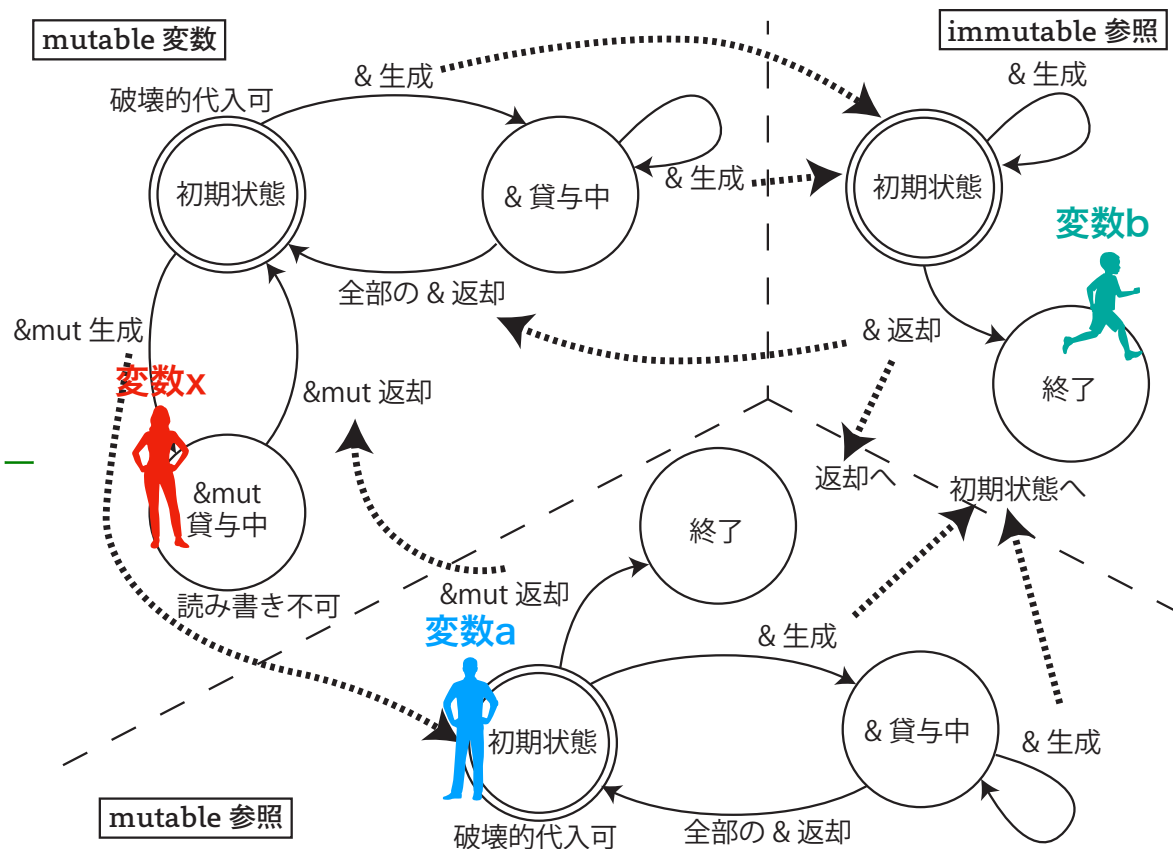
借用の例 (4/5)

```
let mut x = Foo{val: 10}; // xはmutable変数
{
  let a = &mut x; // aはmutable参照
  println!("a.val = {}", a.val);

  // xは「&mut貸与中」状態のためエラー
  // println!("x.val = {}", x.val);

  let b: &Foo = a; // bはimmutable参照
  // a.val = 20;    // aは「&貸与中」状態のためエラー
  println!("b.val = {}", b.val);
  // ここでbが返却される

  a.val = 30;
}
```



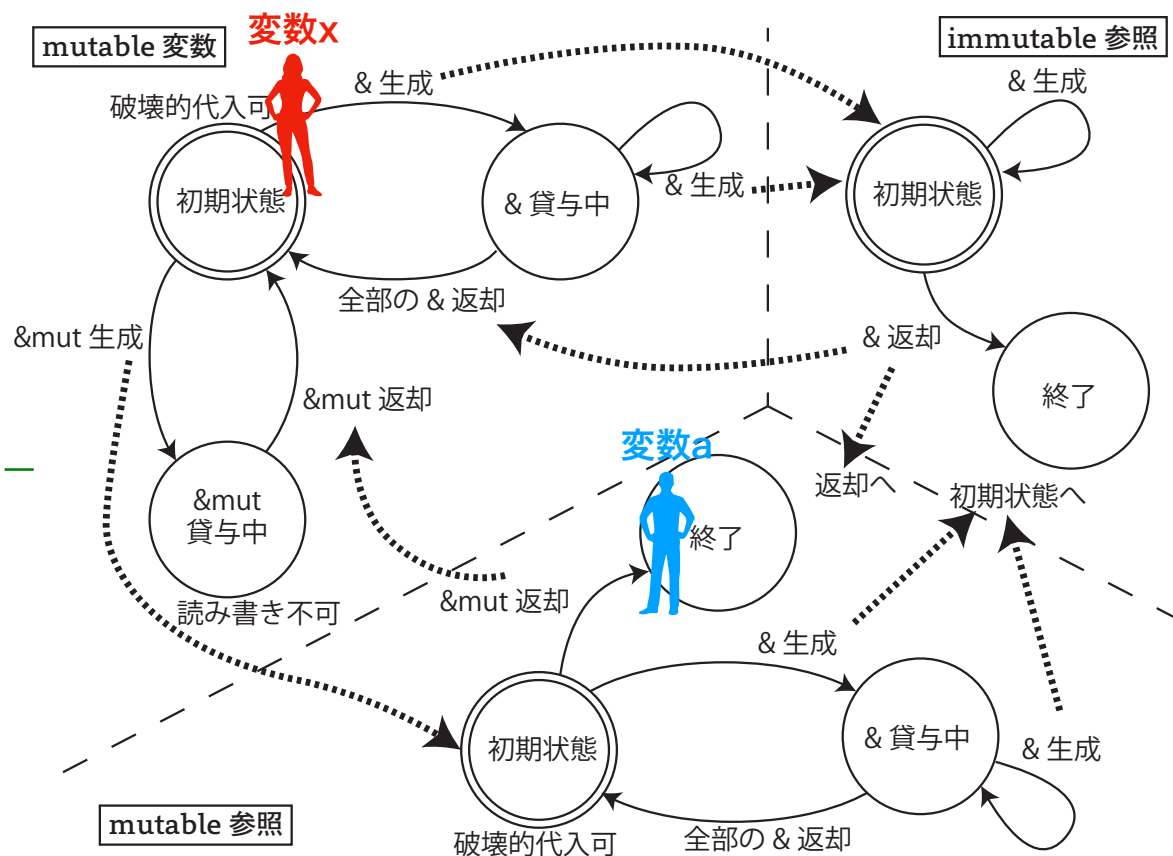
借用の例 (5/5)

```
let mut x = Foo{val: 10}; // xはmutable変数
{
  let a = &mut x; // aはmutable参照
  println!("a.val = {}", a.val);

  // xは「&mut貸与中」状態のためエラー
  // println!("x.val = {}", x.val);

  let b: &Foo = a; // bはimmutable参照
  // a.val = 20;    // aは「&貸与中」状態のためエラー
  println!("b.val = {}", b.val);
  // ここでbが返却される

  a.val = 30;
}
```



レポート課題

レポート課題

- ・簡易版AArch64アセンブリのエミュレータを、Rustで実装せよ
 - ・AArch64とは64ビット版Armのアーキテクチャ
 - ・Nintendo Switch、iPhone、AndroidなどがAArch64
- ・ソースコードは
https://drive.google.com/drive/folders/1N7ceHP0g9z_BkyyCGoLBljS_Tgm7Zyg9?usp=sharing
を利用すること
- ・レポートには、①実装した部分のソースコード(eval.rs)と②(PDFで)解説、実行結果を含めること
- ・簡易版AArch64アセンブリの③入力となるテストコードを自身でも作成し(ex1.Sとex2.Sを倣って、何らかの計算アルゴリズムをex3.Sで作成)テストすること
 - ・なぜ、そのテストコードを採用したのか(②のPDFで)を解説し、実行結果を含めること
 - ・他人と全く同じテストコードの場合、減点する可能性がある
- ・つまり、①②③を提出してください
- ・締め切り：2023年11月29日 23時50分 (JST)

簡易版AArch64アーキテクチャ仕様

- ・ 汎用レジスタはx0からx30のみ
- ・ 命令セットは次のとおり
 - ・ 算術演算命令：add, sub, mul, div
 - ・ 代入命令：mov
 - ・ 比較命令：cmp
 - ・ 条件分岐命令：b.eq, b.lt, b.gt
- ・ 条件分岐のジャンプ先アドレスは行指向で行う（空行は行数に含めない。0オリジン）
- ・ cmp命令では、同じ値（equal）か、小なり（less than）、大なり（greater than）の情報のみをすることとする

算術演算命令

	例	意味
add	add x0, x1, x2	$x0 = x1 + x2$
sub	sub x0, x1, x2	$x0 = x1 - x2$
mul	mul x0, x1, x2	$x0 = x1 * x2$
div	div x0, x1, x2	$x0 = x1 / x2$

代入命令

	例	意味
mov	mov x0, x1	x0 = x1
mov	mov x0, #100	x0 = 100

第2オペランドには、レジスタか即値を指定可能

即値は、「#アラビア数字」と記述

比較と条件分岐命令

	例	意味
b.eq	cmp x0, x1 b.eq #0	if x0 == x1 { goto 0; }
b.lt	cmp x0, x1 b.lt #0	if x0 < x1 { goto 0; }
b.gt	cmp x0, x1 b.gt #0	if x0 > x1 { goto 0; }

アセンブリコードのRust表現

```
#[derive(Debug)]
pub enum Register {
    X0,
    X1,
    X2,
    X3,
    // 省略
}
```

すべてをenumで表現
Register型はX0からX30となるenum型

```
#[derive(Debug)]
pub enum Op {
    Mov(Register, RegOrNum),
    Cmp(Register, Register),
    Arith(ArithOpcode, Register, Register, Register),
    Branch(BranchOpcode, u64),
}
```

Op型はMov、Cmp、ArithOp、BranchOpとなるenum型

オペコードの表現

```
#[derive(Debug)]  
pub enum ArithOpcode {  
    Add,  
    Sub,  
    Mul,  
    Div,  
}
```

```
#[derive(Debug)]  
pub enum BranchOpcode {  
    Beq, // ==  
    Blt, // <  
    Bgt, // >  
}
```

実装方針

- src/eval.rsの中を編集していく(ここを実装)
- run関数の中のloopがメインループ
- ここで1命令ずつ処理を行う
- 命令に応じて、ctx中のレジスタ変数やcond変数を書き換えていく
- eval_cmp関数、Op::Arithの処理の実装ができればOK

```
pub fn run(ops: &Vec<Op>) -> Context {  
    // レジスタの初期化  
    let mut ctx = Context::new();  
  
    let mut pc = 0; // プログラムカウンタ  
    loop {  
        // オペコードの種類によって実行する処理を切り替える  
        match &ops[pc] {  
            Op::Mov(dst, src) => {  
                eval_mov(&mut ctx, dst, src);  
            }  
            Op::Cmp(reg1, reg2) => {  
                eval_cmp(&mut ctx, reg1, reg2);  
            }  
            Op::Arith(opcode, reg1, reg2, reg3) => {  
                // ここを実装  
                // 次のような関数を定義して実装せよ  
                // eval_arith(&mut ctx, opcode, reg1, reg2, reg3);  
            }  
            Op::Branch(opcode, line) => {  
                if eval_branch(&ctx, opcode) {  
                    pc = *line as usize;  
                    continue;  
                }  
            }  
        }  
    }  
}
```

レジスタ変数

レジスタ変数は以下のContext型に保持
つまり、CPUの持っている情報

```
// レジスタ
#[derive(Debug)]
pub struct Context {
    pub cond: Condition, // cmp命令実行後の結果を保存するレジスタ
    pub x0: u64,
    pub x1: u64,
    pub x2: u64,
    pub x3: u64,

    //省略
}
```

実行例：加算

```
$ cargo run ./example/ex1.S  
result: Context {  
    cond: Eq,  
    x0: 4,  
    x1: 1,  
    x2: 3,
```

省略

x0が4になっていれば成功

入力ファイル (ex1.S)

```
mov x1, #1  
mov x2, #3  
add x0, x1, x2
```


実行例：階乗


```
$ cargo run ./example/ex2.S
result: Context {
  cond: Eq,
  x0: 362880,
  x1: 10,
  x2: 1,
  x3: 10,
```

省略

x0が10! = 362880になっていれば成功

入力ファイル (ex2.S)

```
mov x1, #1
mov x0, #1
mov x2, #1
mov x3, #10
mul x0, x1, x0
add x1, x1, x2
cmp x1, x3
b.lt #4
```



条件分岐命令は、行指向なので、この場合、上から4番め（0オリジン）の命令へジャンプ