第１１回ＵＥＣ杯コンピュータ囲碁大会

きふわらべ
アピール文書

２０１９年１２月０１日　高橋智史

TUYOSA
強さ　に影響しない☆

DARE
誰なのよ？

開発者
高橋　智史

「コンピューター囲碁の　思考部　の大会なのに
ジー　ユー　アイ
ＧＵＩ　ばっかり作っている……☆」

北白河ちゆり
／TOHO PROJECT
FANMADE　※

コンピュータ将棋エンジン
きふわらべ
モンテカルロ　ツリー　サーチ
「ＭＣ　Ｔ　Ｓ　の
アッパー　コンフィデンス　バウンド　ワン
Ｕ　Ｃ　Ｂ　１　だけだろ☆　説明終わりだぜ☆」

「ラムダ計算の話しを１００ページぐらい書くんでしょ！
提出期限まで　もう１１時間ぐらいしかないわよ！」

岡崎夢美
／TOHO PROJECT
FANMADE　※

※北白河ちゆり、岡崎夢美は　東方夢時空　の登場キャラクター／

LAMBDA CALCULS

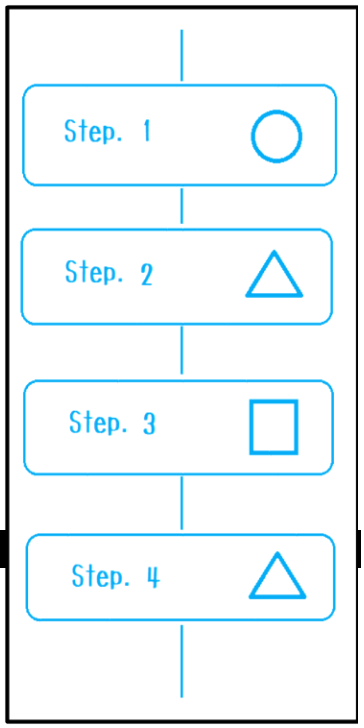
ラムダ　計算　って何なの？

HIKISU　KAERICHI

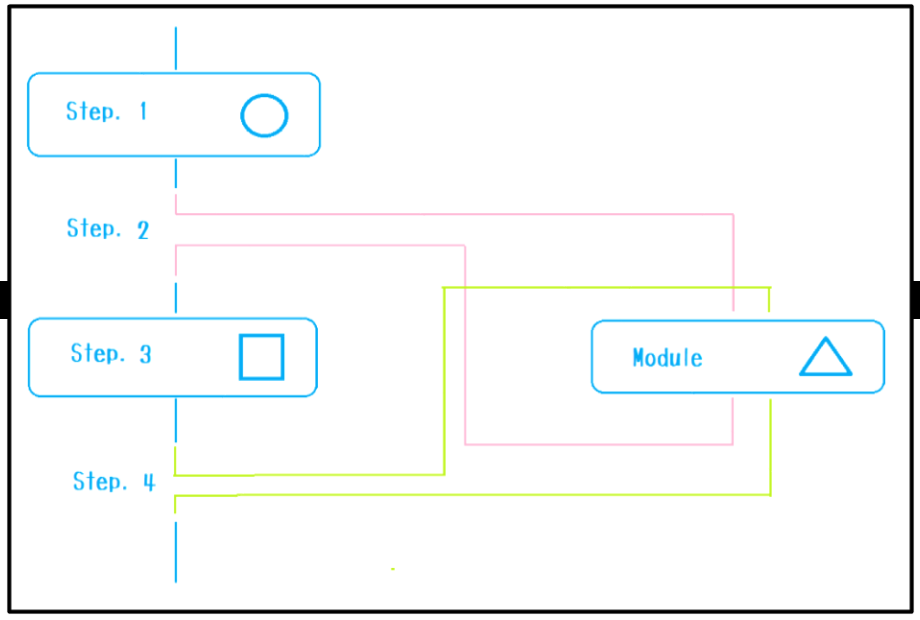
引数　や　返り値　に関数が使えるやつだぜ☆

「ステップが　フラット　に並んでいるときは→
かんすうがた
関数型プログラミング　も　そうでないものも
何も変わらない……☆」

「フラット　に並んでいない　プログラム　が
あるのかだぜ☆？」



Sequence

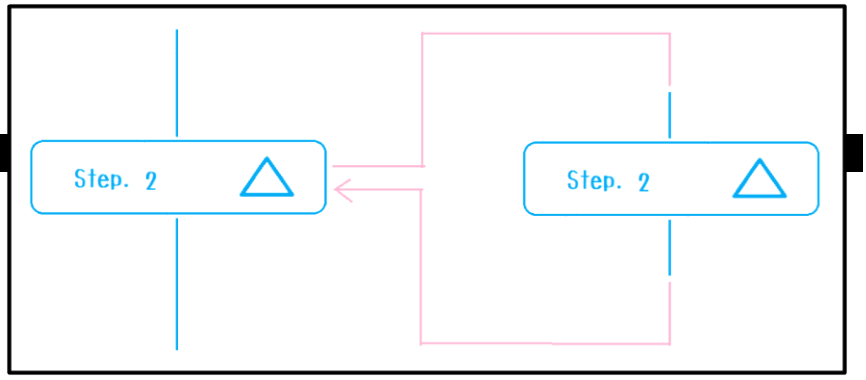


Sequence, too

「↑よく使う　ステップ　を外に切り出し、
どこからでも　呼べるようにすると
フラットではなくなる☆」

「サブ・ルーチンよね」

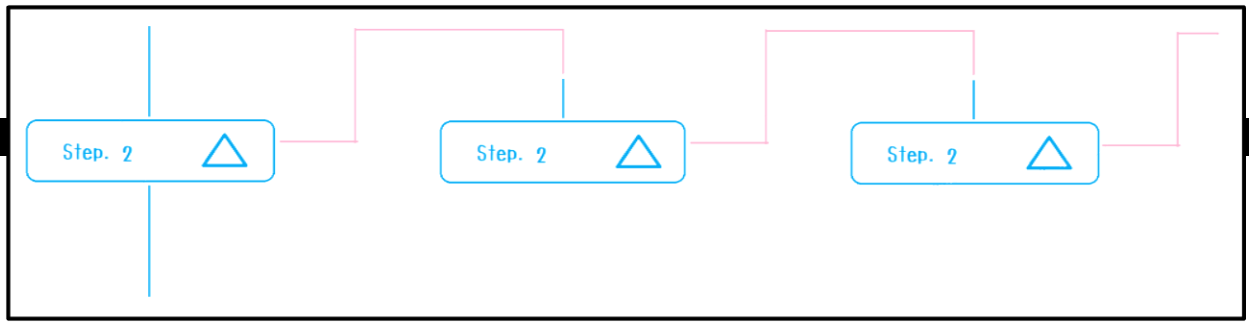
First-order function, Recursive



「↑じゃあ　ステップ２　が自分自身を　呼び出したら
どうなるんだぜ☆？」

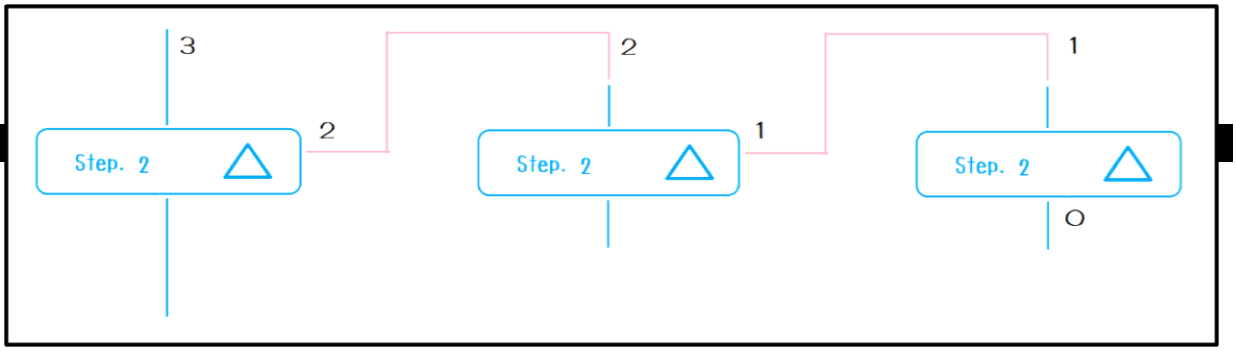
「そんなことをしたら　いかんのでは……☆？」

「これは上図のようには　ならないのよ」

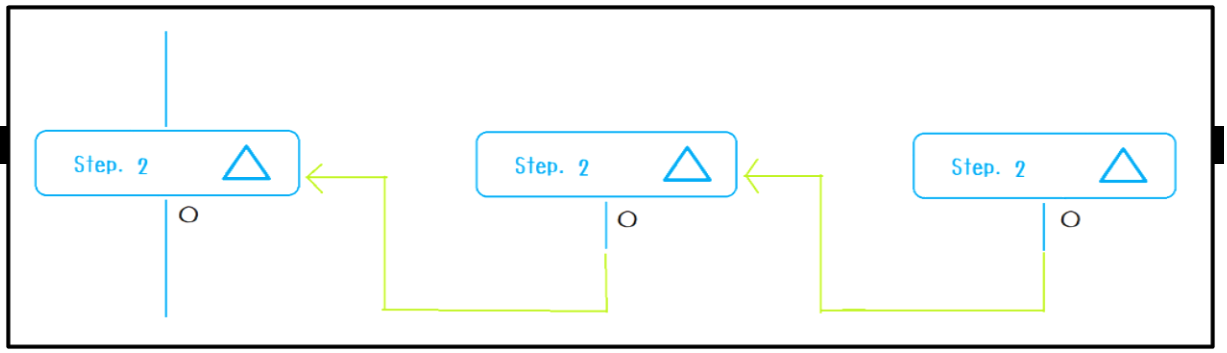


「↑帰りが無いんだから」

「帰ってこようと思ったら　カウンターのようなもの　が要るよな☆」



「 ↑例えば　入ってくる数　が　0より大きい　ときは　自分を呼び出して……☆」



「 ↑0になったら　自分の呼び出し元に帰るんだぜ☆」

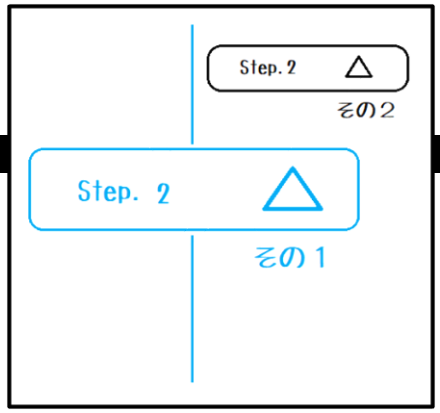
「 一階(いっかい)　の再帰関数だな☆」

「 ここは　一階　だったのかだぜ☆？」

「 わざわざ　もったいぶって　関数型プログラミング　というときは、
二階以上(にかいいじょう)　の関数を使うことを言う☆」

「 わらちゃんの説明で　合ってるじゃない！
いちゃもんよ　いちゃもん！
再帰するのに　二階は　しゃしゃり出てこなくていいのよ！」

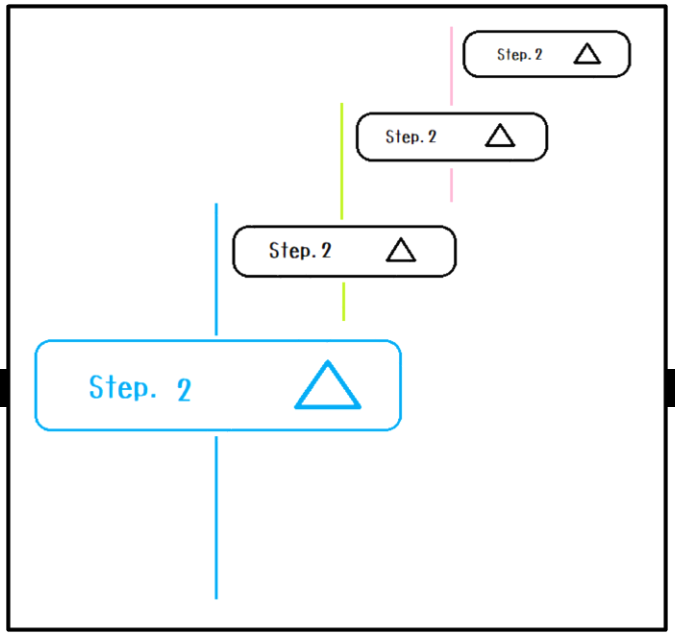
Second-order function goes into a First-order function.



「ステップ２（その１）　の中に→
ステップ２（その２）　を入れて、
ステップ２（その１）　は、
ステップ２（その２）　を呼び出したらいい☆
これで　深さ１つの再帰関数と同じだぜ☆」

「何言ってるか　分かんないのよ！」

Fourth-order function goes into …

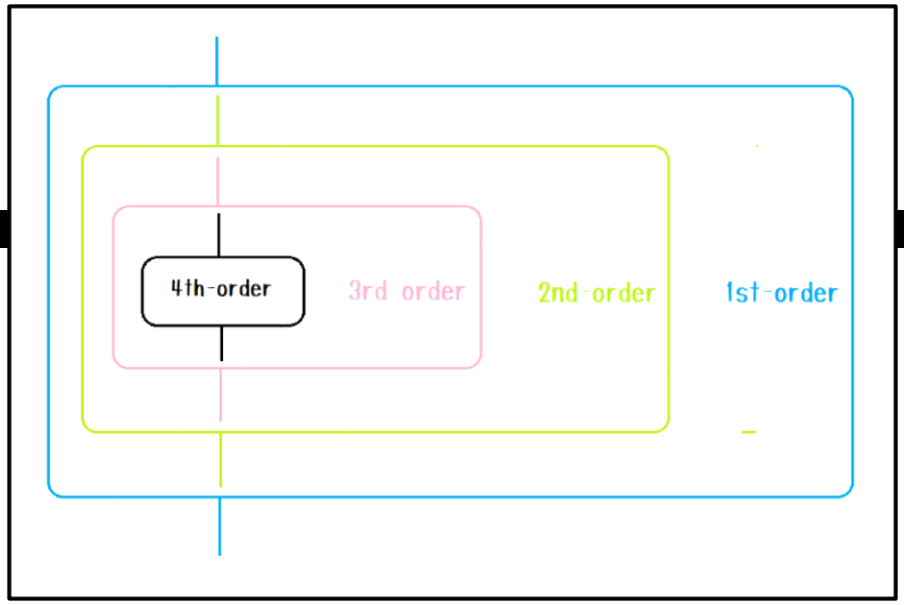


「つまり　４階　の関数を　３階　に入れて↑
４階の関数が入った３階の関数　を　２階　に入れて
４階から２階の関数が入った関数　を　１階　に入れれば
カウンターのようなもの　は　不要　☆」

「そんなところに　張り合われても……☆」

「図のシーケンスが　切れてない？
どう　つながってんの？」

Higher-order function



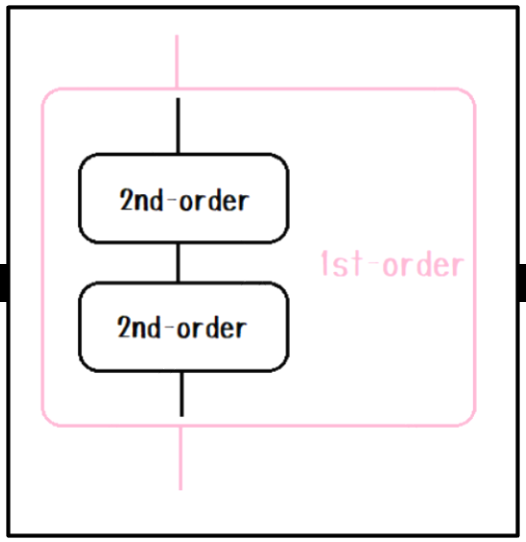
「 切れてなんか　ないぜ☆

上図は　４階関数、または　高階関数（こうかい）　と呼ぶぜ☆」

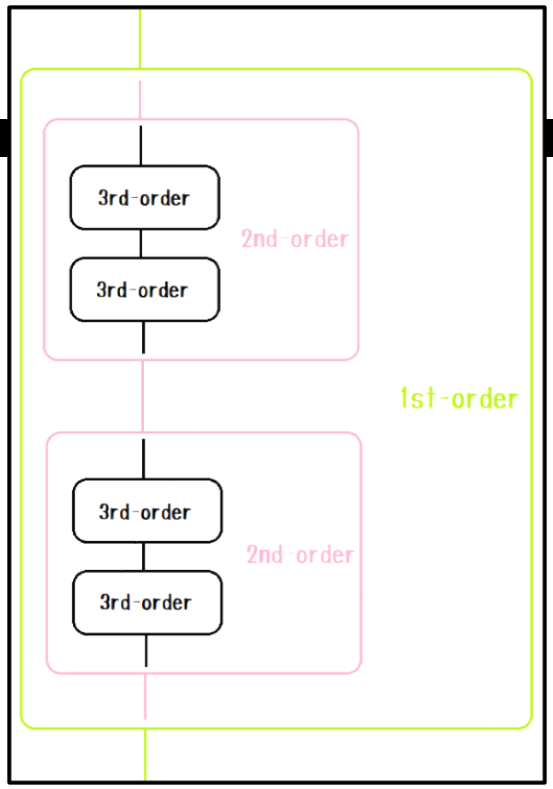
「 再帰する回数分　あらかじめ
関数を　関数の中に　プッシュ　したのかだぜ☆」

「 この　高階関数　が　どのような順で実行されていくのかを
LAMBDA CALCULS
手計算で体験できるのが　ラムダ　計算　だぜ☆」

「 これ以上どう問題が発展しようがあるの？」



「 ↑２階関数が　２回　呼び出される１階関数　に　自分を入れると☆」



「 倍々ゲームが始まるぜ☆」

「 嬉しくない……☆」

「 囲碁も将棋も
５００手も指さないんだから
自分を自分に
５００回プッシュする一手指す関数があれば
囲碁プログラムできんじゃないの？」

「 再帰でいいよな☆」

「 ラムダ計算の　何が嬉しいのか　分からん……☆」

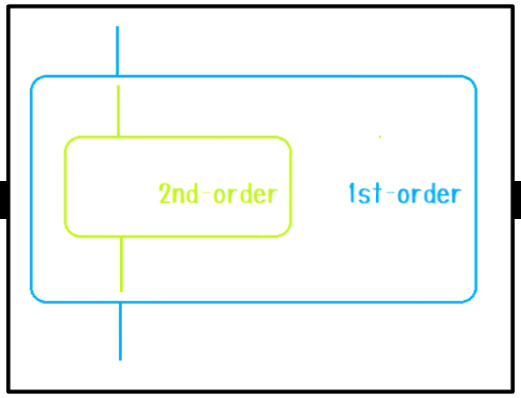
「 ラムダ計算　自体は
計算を作るフレームワーク　であって、
数の　１　があるのなら　１　を使った方が楽だし、
足し算の記号　＋　があるのなら　＋　を使った方が楽だし、
二乗の／ーテーション（記法）があるなら　／ーテーションを使った方が楽だし、
関数ｆ　があるのなら　関数ｆ　を使った方が楽だぜ☆」

「つまり
数が無くても、　足し算の記号が無くても、　二乗のノーテーションが無くても、
関数ｆ　が無くても、
高階関数があれば　同じことができると言っているのが　ラムダ計算　だぜ☆」

「ラムダ計算したくないんだけど」

「チューリング・マシンを　高階関数にしただけか……☆」

「ラムダ計算の初期セットに　自然数　もないの　うんざりなんだけど！」



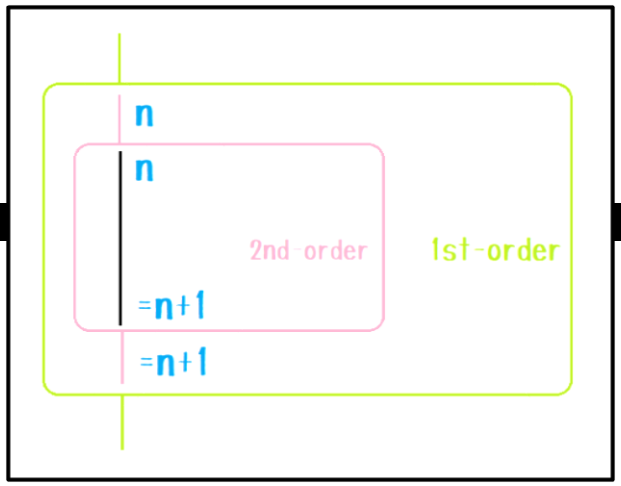
「↑自然数の　２　は　２階関数　なのよ？」

「どういうことだぜ☆？」

Zero and `n+1`

「プラス　と　イチ　も　本当は　高階関数　を使って作るんだけど
今は　もう作ったことにしましょう。計算が無かったことでゼロを表すとします。
ゼロと　ｎ＋１　を使って　自然数２　を作ってみましょう！」

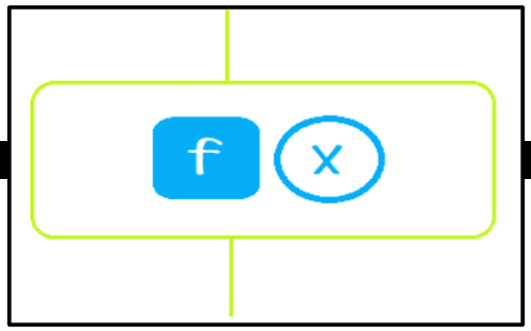
`+2`



「↑こういう関数でもあれば　０　を入れれば　２　が出てくるが☆」

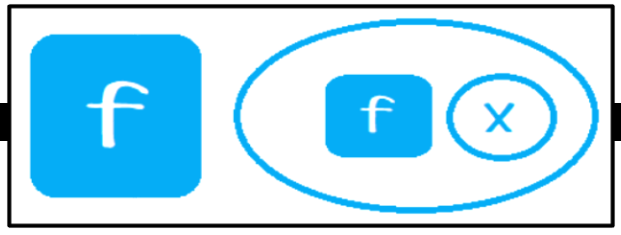
「その図の書き方だと　あんまり　高階関数　に見えないんだぜ☆
どういうことかと言うと……☆」

`ｘをｆします`



「`ｘをｆする`　という無意味な式があったとしよう☆
ｆ　は　あとで　`＋１`　を入れるつもりだから　むずかしくないだろう☆」

Push self into x.

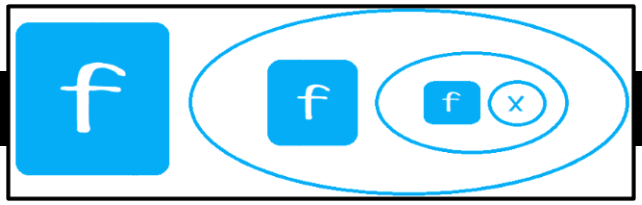


「`ｆ（ｘ）`　のｘに自分自身を入れると　`ｆ（ｆ（ｘ））`　になる☆」

「ＢＡＳＩＣ　や　Ｃ　などのプログラム言語は　引数に数を入れるけど、
関数型プログラミングは　関数　を入れるのね」

「 プログラム言語では　コールバック　というやり方だな☆」

```
`f(x)` push `f(f(x))`
into x.
```

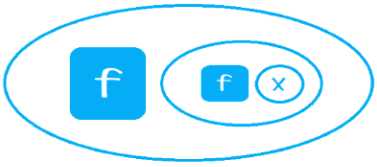


「 ↑もういっちょ　プッシュ☆」

「 なんてことを　してくれるんだぜ☆」

```
Psuedo code
```

```
(f,  x)  =>
{
}
```



「 ↑今どきの　名無し関数　の書き方なら　こんな感じになるかだぜ☆
呼出し側で　どうやるかも　見ておこう☆」

```
Psuedo code
```

```
callback(`=?+1`,  0)
```

```
callback(s=>s+1,  0)
```

「 ↑プログラミングでは　上の疑似コードの桃色の部分は
クロージャ　と呼ばれたりするな☆
ラムダ計算では　アブストラクション（ラムダ抽象）　と呼ばれる箇所だぜ☆」

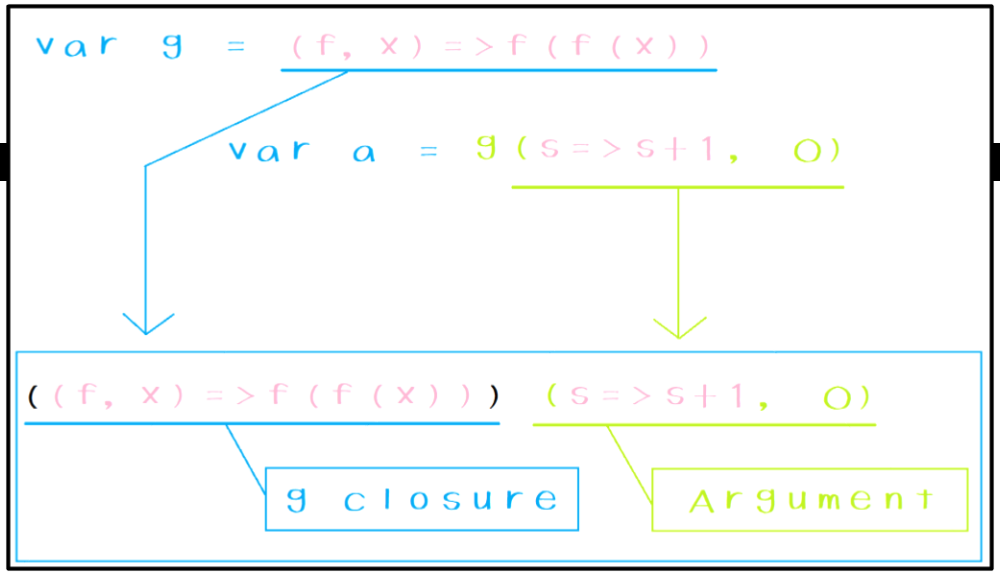
「 クロージャ　の　s　には何が入ってんの？」

「プログラマーが　ひねった使い方をしていないなら
計算結果　を入れてるはずだぜ☆
試しに　手計算　をお見せしよう☆」

Psuedo code

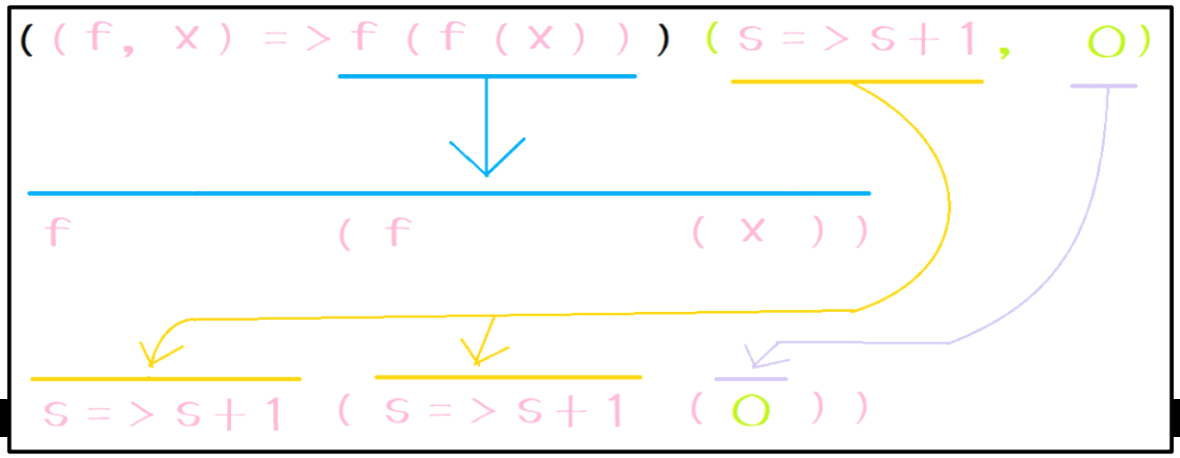
```
var g = (f, x) => f(f(x))
var a = g(s => s+1, 0)
print(a)
```

「↑上図の　桃色が　クロージャ、　黄緑色が　関数コール　だぜ☆
頭に　`引数 = >`　が付いてたら　クロージャ　と見分けろだぜ☆」

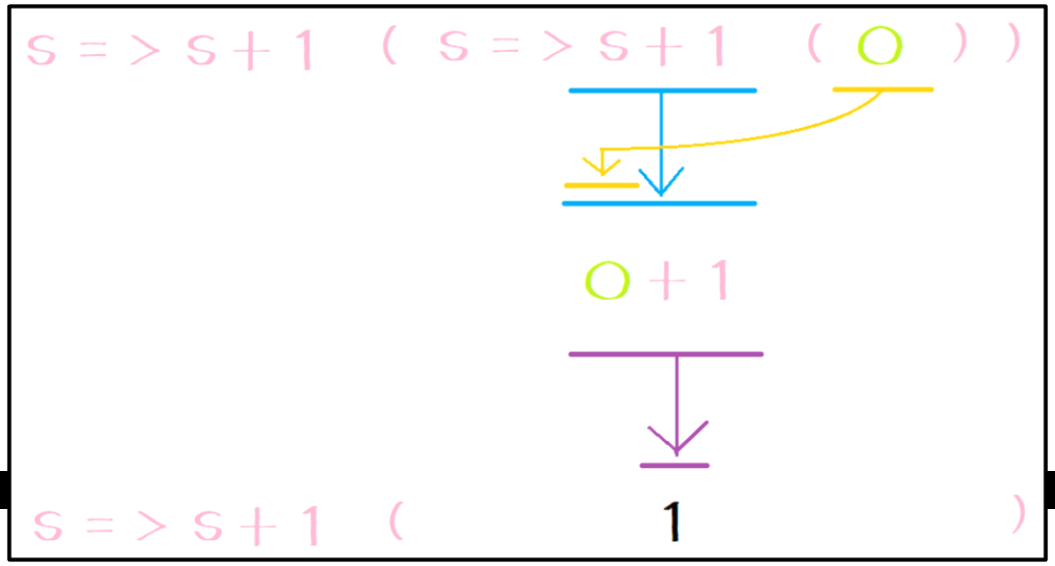


「g　に式本体を代入すると　↑上図　のように
プログラム言語で言うところの　関数コール文　になるが、
これは　ラムダ計算　では
アプリケーション（ラムダ適用）　と呼ぶものだぜ☆」

「どう計算するんだぜ、こんなもん☆」

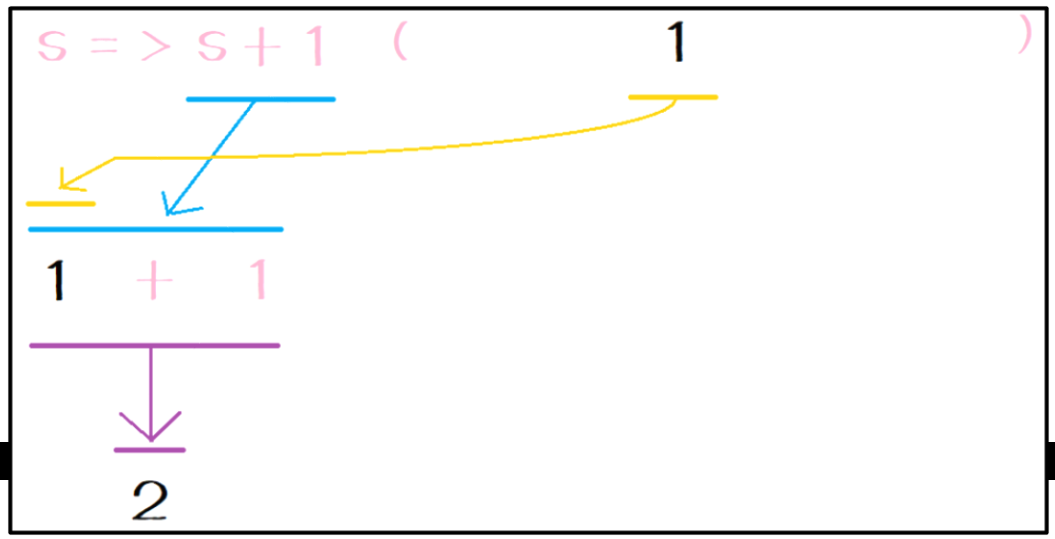


「 引数の並びをよく見て　代入するだけだぜ☆」



「 アプリケーションがあれば　代入して　計算を進めていっているが、
ラムダ計算では　代入のことを　βリダクション（ベータ簡約）　と呼ぶ☆」

「 ヒマなんだな……☆」



「 ほら　２　が出た☆
｀ｆ（ｆ（ｘ））｀　と　｀（ｆ，ｘ）＝＞ｘ＋１｀　があれば
２　は作れるぜ☆
ペアノの公理　と　自然数の２　が　ラムダ計算　で紐づいた瞬間だぜ☆」

「 自然数の２　を、　アラビア数字の２　を使わずに　計算で出したの？」

「 そう☆！」

「 やっぱり　ラムダ計算するやつは　ヒマなんだな……☆」

CLOSURE

クロージャー　をカンペキに理解しようぜ☆？

TOCHU

プログラムの　途中　で関数が定義できるせいなのよ。

「 １１月３０日は　あと　４時間もないから
ラムダ計算　の話しで
他のことを押しのけてでも　押さえておきたい　のが　クロージャー　だぜ☆」

「 ＰＲ文書なんか　書いてないで
コンピューター囲碁プログラム書けばいいのに……☆」

「 関数型プログラミングじゃない言語は、
関数定義文は　プログラムの途中じゃないところに　書いてあるものなのよ」

Global scope

「 例えば　ファイルの
トップ・レベルだったり……」

```
void connect(url, port)
{
  ...
}
```

```
class Logger
{
    writeLn(message)
    {
        ...
    }
}
```

In the class

「 クラスの中に　入ったのよ」

「 関数には　名前　が付いてるものだったよな☆」

「 関数を　呼ぶ　のだから、　名前　は要るだろ☆」

```
go(one)
{
    var callF = (f, two)=> {
        var four f(two);
        return four;
    };

    var five = callF(three=>{three+1}, one);
    return five;
}
```

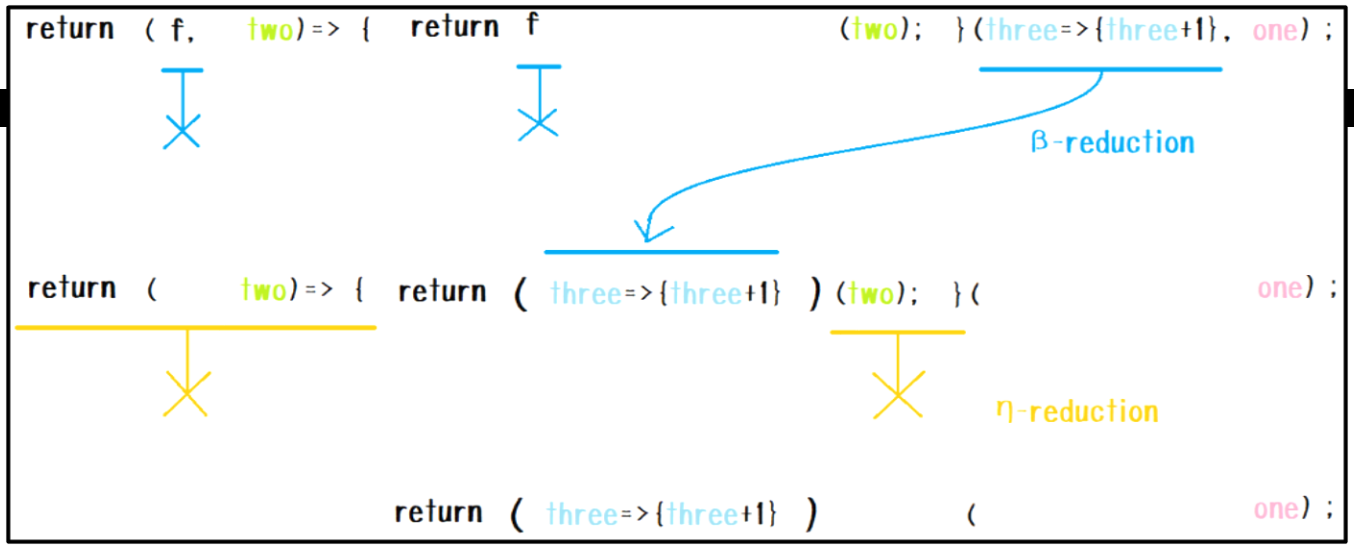
「 良いサンプルを即座に思い浮かべられなかったが、
`var callF` とか　わざわざ　名前の付いた入れ物に　関数を入れなくても、
first-class object
関数を　第一級オブジェクト　として扱う言語なら、
名無し関数　を　引数に直接書ける☆」

```
go(one)
{
      return  (f, two)=> {
          return  f(two);
      }(three=>{three+1}, one) ;
}
```

「 アプリケーションの書き方にしても　見づらく思わないなら、
アブストラクションの後ろに　引数をくっつけてもいいだろう☆」

「 そのサンプル・コードは　η簡約（イータかんやく）　できるだろ☆」

「 ☆？」



「 引数を　バケツ・リレー　してるだけの皮　に意味はないから
べりっと　剥がして……☆」

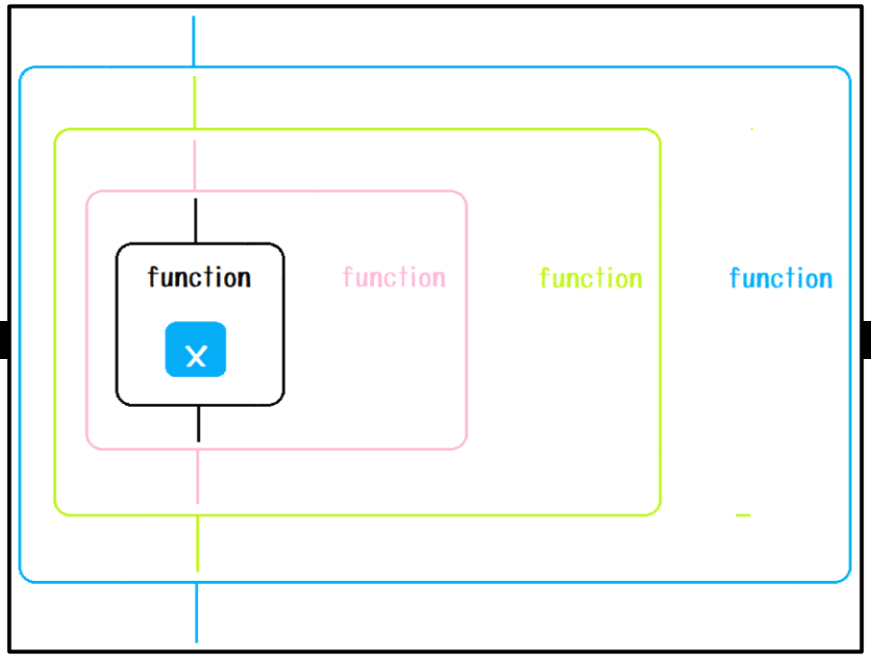
```
go(one)
{
        return   ( three=>{three+1} ) ( one) ;
}
```

「 ↑こうだぜ☆」

「 くそっ☆　覚えてきたな☆」

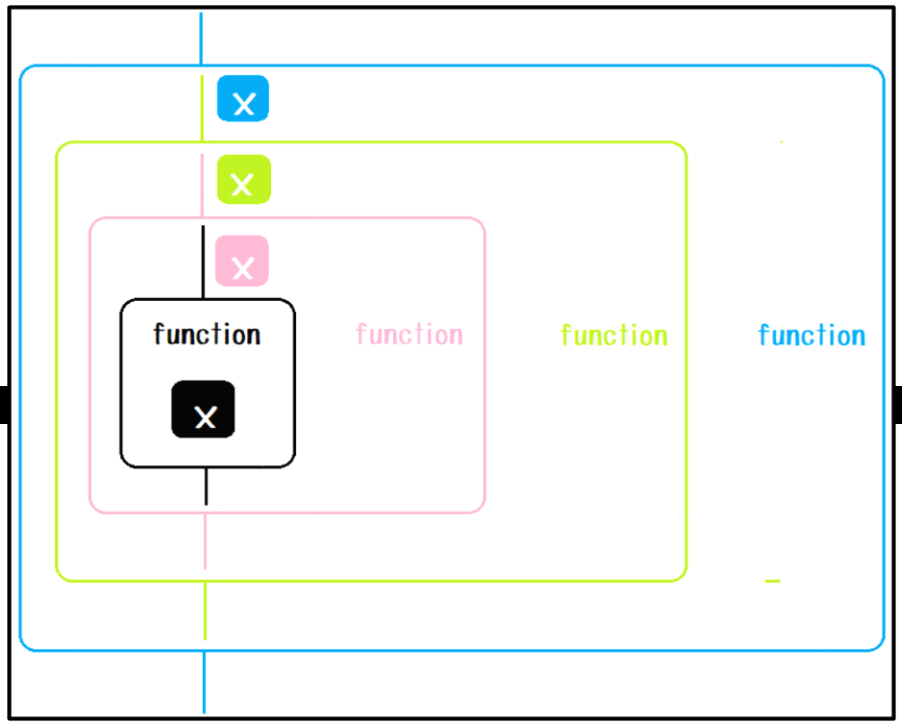
「 簡約できるものを　わざわざ　ラムダ式　にする必要なくない？」

「 ラムダ式　で書いていると　ビターッ　と隙のないコードを書くようになるぜ☆
グローバル変数を使う必要性が　どんどん　無くなる☆」



「 ↑例えば　深いところ、　木でいうと　根から見て　葉のところで、
根のような浅いところで設定した変数を　使いたいことがある☆」

Bound variable



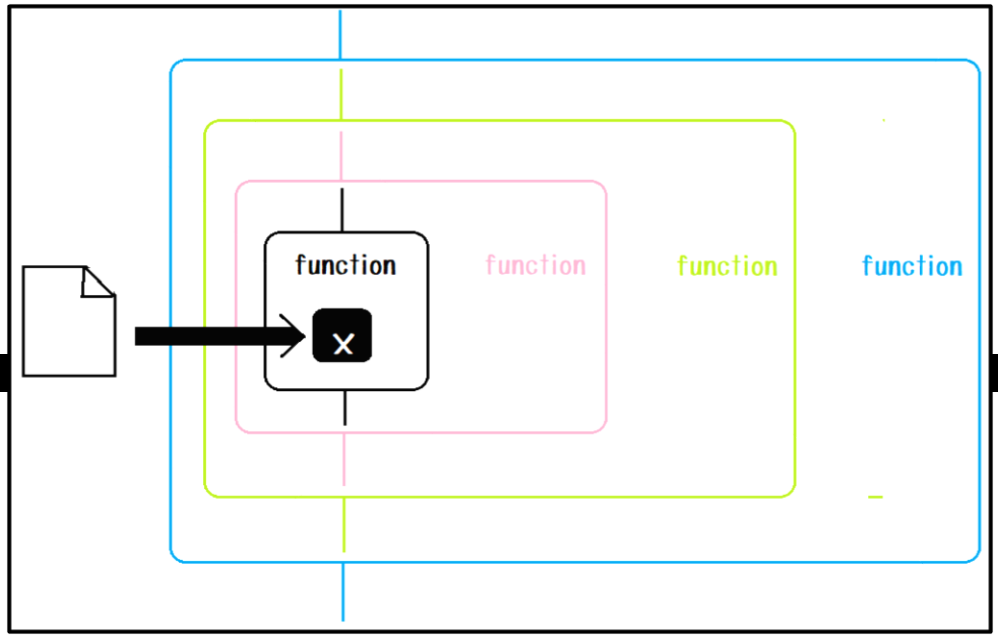
「 ↑そんなときは　例えば
function青（ｘ）、function緑（ｘ）、function赤（ｘ）、function黒（ｘ）
のように　関数の引数で　バケツリレー　してもいい☆」

「 それは設計が　おかしいんじゃない？
行きつくところ、関数の　引数　に　すべての変数　を並べて回るの？」

「 定数　π（パイ）　を　いつ使いたくなるか　分からないだろ☆
すべての関数に　引数　π　を　付けておくのかだぜ☆？」

「 定数は　グローバルなスコープに　置いてあっても　困らないだろ……☆
イミュータブル（不変）　だからな☆
困るのは　ミュータブル（可変）　な引数だぜ☆」

Free variable



「また別な方法としては、外部ファイルから読み込んだり、
グローバル・変数を　読み取ったり　するのもある☆

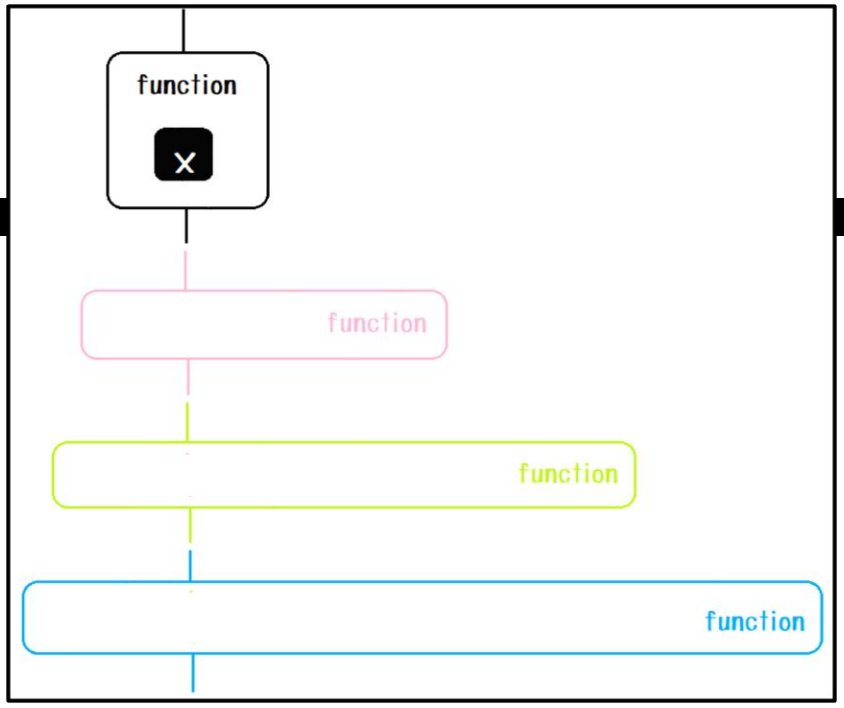
STATEFUL

しかし　これは　内部状態を持つ　ようになり、
外部ファイルの設定を　いじったことを忘れていて
昨日と、今日で、動きが異なる、なんてこともあり　テスト　が困難になる☆」

「それも　やっぱり　設計がおかしいのよ。
葉っぱの１つ１つで　異なる設定ができる　便利さ　というのは、
葉っぱの１つ１つで　異なる設定をしなければならない　不便さ　なのよ」

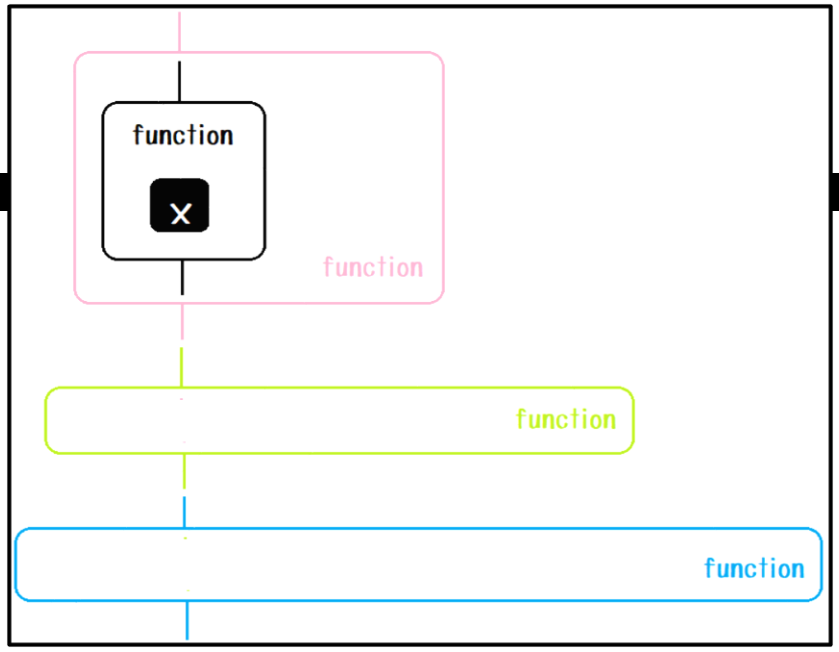
「プログラムの開始時に　状態はいつも　同じであってほしぜ☆
異なるなら　開始時に　チェックしたいぜ☆」

Free variable, Context



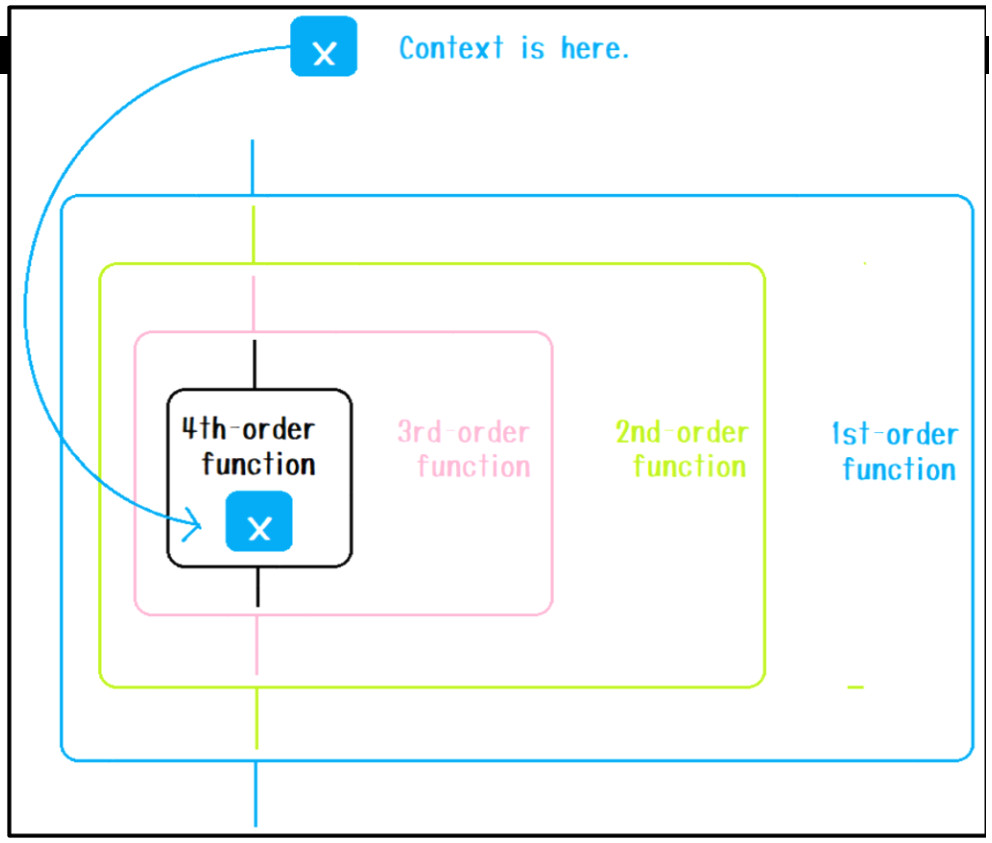
「関数型プログラミングなら、　葉っぱに近い方の関数　から先に作って
幹に近い方の関数　に渡す、ということができる☆」

Free variable, Context



「このときの　変数ｘ　は、その関数が作られたスコープ　に従属している☆
このことを、
変数ｘは　関数が実行されるところ　ではなく、
定義した　ところ（コンテキスト）　に縛られている、　と表現したりするぜ☆」

Free variable, Context

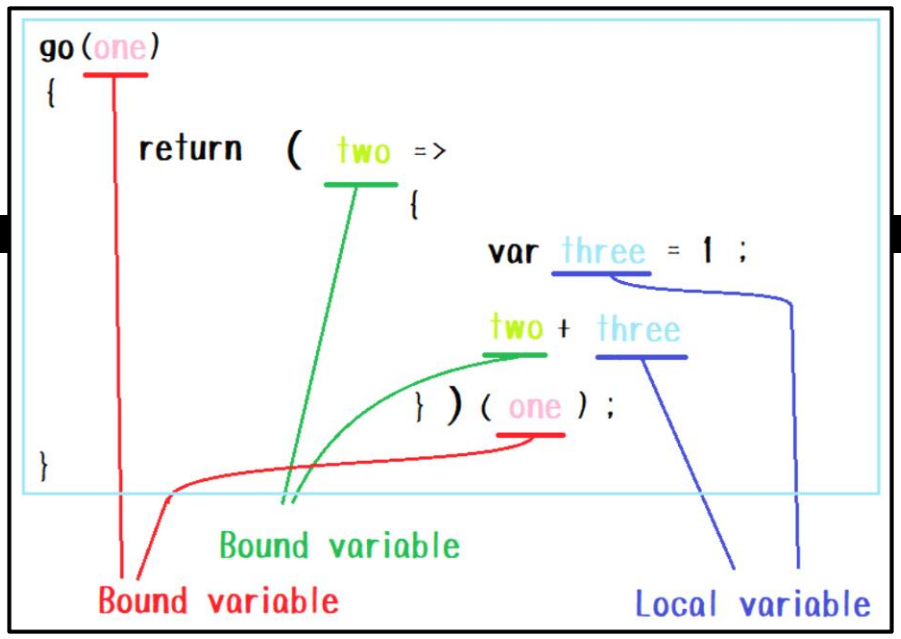


「 だから　高階関数　を使えるお陰で
巻き寿司の一番内側のタマゴ　を最初に入れておく、
ような逆転現象が　可能だぜ☆」

「 その変数xって　どうやって　入れてんの？
引数　以外に　入れるところがあるの？」

「 ある☆」

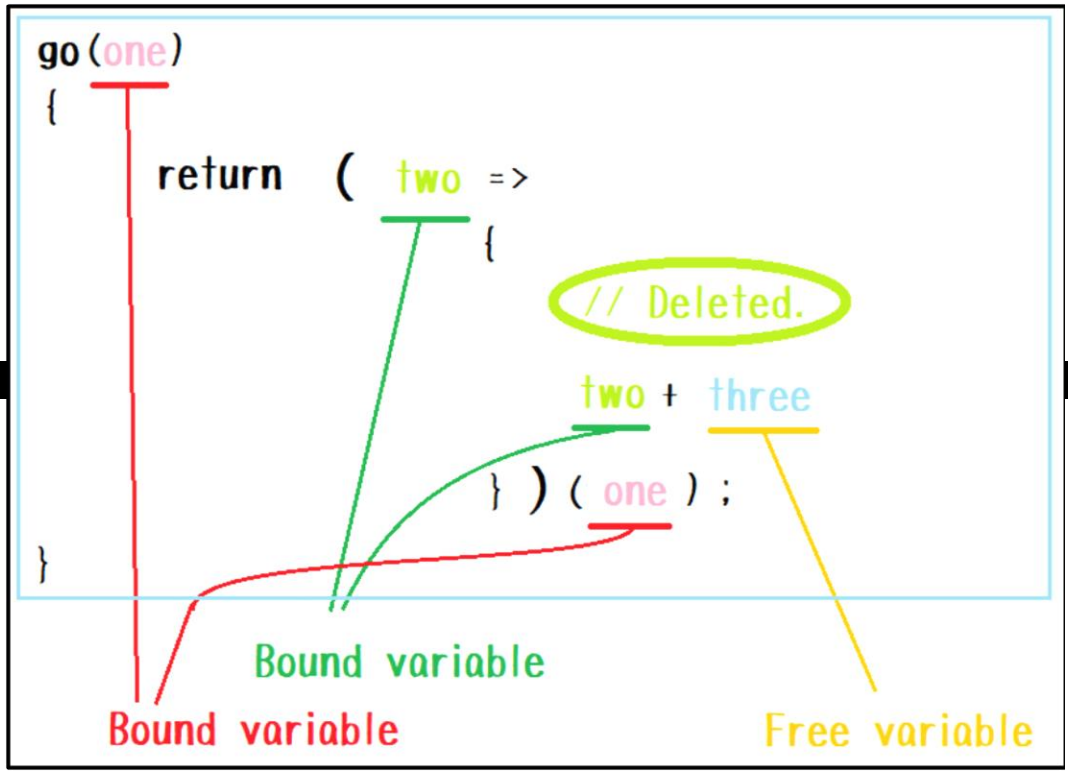
Bound variable



「 実行可能なプログラムは　変数はすべて、
Bound variable　　Local variable
束縛変数　　か、　　ローカル変数　　のどちらかだぜ☆」

「 グローバル変数も　あるんじゃないのか☆？」

Free variable



「確かに `var three = 1;` を消すと three は未定義な変数になるが、
もしかすると グローバル変数かもしれない☆
Free variable
このとき three は、 自由変数 だぜ☆」

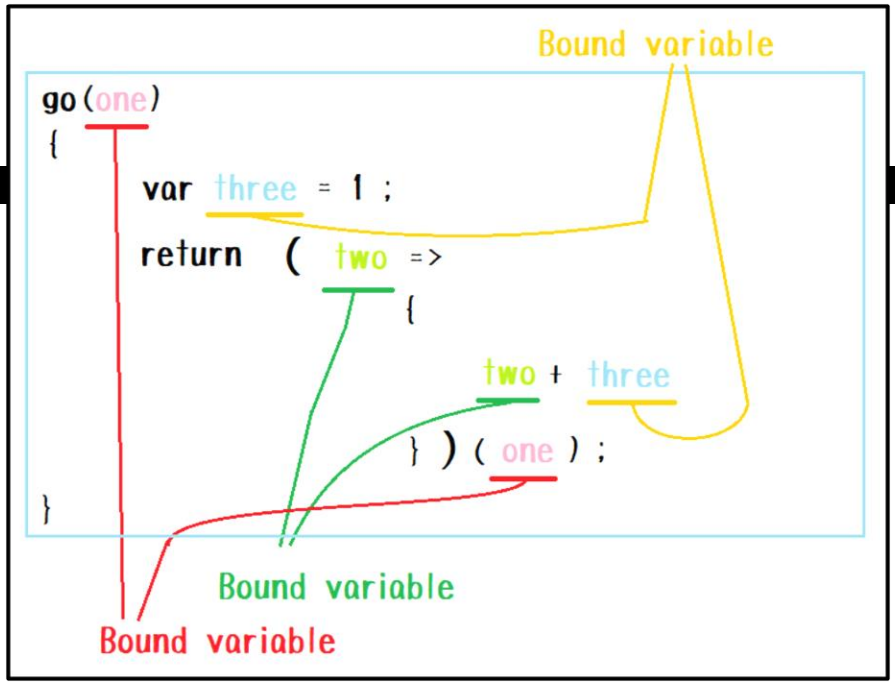
「自由変数があると three を unknown にするか、
プログラムが実行不能になるか なんだけど」

「自由変数は 関数の外側 から飛び込める変数かだぜ☆
そんなもんが あったんだな☆ 気にしてなかったぜ☆」

「関数型プログラミングではない言語なら、
自由変数は クラスのプロパティ だったり、
別モジュールからインポートした定数 だったり、
グローバル変数 だったり するだろ☆」

「このとき コンテキストは クラス だったり、
別モジュール だったり、 グローバル・スコープ だったり
するわけだが……☆」

Bound variable



「関数型プログラミングでは
Enclosure
外側の関数　というコンテキストもあるわけだぜ☆」

「関数が　あたかも　プロパティのように　変数を持てるなんて
不思議ねぇ。　こんなの　ありなのかしら？」

「まずは　理解しろだぜ☆
次に　やりまくれだぜ☆」

RETURN

関数も　返せる　んですって？

LAZY EVALUATION

遅延評価　　が可能になるんだぜ☆

「まだ説明してなかったが……☆」

```
go(one)
{
        // 関数の実行結果を返すぜ☆（＾～＾）
        return  ( three=>{three+1} )( one);
}
```

```
go(    )
{
        // 関数を返すぜ☆（＾～＾）
        return      three=>{three+1}          ;
}
```

Retuan a function

「関数は　関数の実行結果を返すだけではなく、
関数そのもの　を返す　こともできるぜ☆」

「なんで　そんなことをするの？」

「あとで　値をもらえることは確実なんだが、今もらうのは　まだ得策ではない、
みたいなときに　関数をもらっておく　んだぜ☆
値が欲しくなったときに　関数を実行する　という時間の融通が利くぜ☆」

「そんな重要そうなことを　なんで今になって　しゃべるんだぜ☆？
１１月３０日は　あと　１３分しかないぜ☆？」

「非同期処理で　別のコアで計算した結果を取得する関数　を　もらっておき、
定期的に　その関数を　せっつきながら
値が取れたときに　集計する、　みたいな使い方が考えられるな☆」

「マルチ・コア・プログラミング　が流行りだしたら
関数型プログラミングは　必須のスキルになると思うぜ☆」

「はい時間！　もう無理！　まとめに入りましょう！」

Y COMBINATOR

Yコンビネーター　の説明もできないなんて……☆

ジャバ　スクリプト

Java Script　でもやってりゃ自然に覚えるわよ！

「 .map( ) とか使い始めると .forEach( ) も使わなくなるよな☆」

「 お父ん、時間切れだぜ、まとめろ☆」

「 Python 使ってても C# 使ってても
ラムダ計算由来の　メソッド・チェーン　とか　やりまくるし、
Rust とか　関数型プログラミングを初めてやるときに出すバグを
言語仕様で防いでいるよな☆
特に Null を返せないのは　初歩的エラーを　大幅に減らしているところで、
if～else文も 関数のように値を返せるのが
値を一時的に格納するだけの変数を減らせて ウマい☆」

「 関数型プログラミングからお金をもらってるのか、というぐらい
ハマってるわよね」

「 きふわらべ　の強さには　一切　関わらないけどな☆」

「 くそっ☆！」

>>> Dad! When will we learn functional programming and aim for machine learning multicore programming?!