

# Parsec による簡易インタプリタの実装と構文設計 (最終発表)

機械科学・航空宇宙学科 3 年 1w192224 田久健人

August 2, 2021

# はじめに

中間発表の続きです...

今回は、パーサではなく評価寄りの話です

進捗→ [https://github.com/tkyawa/project\\_research\\_a](https://github.com/tkyawa/project_research_a)

# abstract

Haskell のパーサコンビネータの Parsec を用いた簡易なインタプリタの実装を通して、パーサコンビネータの雰囲気を感じ、実際に構文の設計を行う。

# 目次

- ① 関数実装に向けた下準備
  - おさらい
  - Return 文の実装
- ② 関数の実装
  - 関数型の検討案①
- ③ 再帰呼び出し実装の検討
  - 再帰呼び出し実装における問題点
  - 案②: スタックベースの環境
  - スタックベースのスキープの挙動
- ④ おまけ
- ⑤ 感想, その他

# これから拡張していく文法

AST→

```
data Statement = Seq [Statement]
                | If Expr Statement Statement
                | While Expr Statement
                | Assign String Expr
                | Skip
```

```
data Expr = Var String
           | Integer Integer
           | Bool Bool
           | (Op) Expr Expr
```

# これから拡張していく文法

(実際の構文 →)

```
Statement ::= Statement; Statement; ...,; Statement
            | if(Expr)then{Statement}else{Statement}
            | while(Expr)do{Statement}
            | String := Expr
            | skip
```

# これから拡張していく文法

実行時の値を保存する環境 (実行コンテキスト), 型の情報

```
type Env = Map String TypeEnv
```

```
data TypeEnv = TypeInteger Integer  
              | TypeBool Bool  
              deriving(Show)
```

# 関数実装に必要なもの

関数の構文 →

AST:

```
data Expr = ...  
          | Func String Statement
```

(実際の構文:)

```
Func ::= func(arg){body}
```

関数は式だが中身が文 → 値 (式) を返す文が必要



# Return 文の実装

文を, (形式上は) 値を返すようにする (evalStatement の型を統一)  
→Null 型の実装を考える

```
data TypeEnv = ...  
              | Null
```

return 以外の文は, Null を返すようにする

## Return による jump 機能の実装①

一般的な Return 文・・・jump の機能がある

例:C 言語の場合

```
int hoge()  
{  
    int temp = 1;  
    return temp;  
    temp = 2;  
}
```

hoge() を呼ぶと, 1 が返ってくる →return で文の評価が終了している

## (補足)Return による jump 以外での同等の表現

Rust:

body の最後に式を書くことを許し、それが関数の返り値となる  
return で処理を中断して値返すことも可能 (jump)

OCaml (関数型言語一般):

そもそも全部 Expr

必ずしも return による jump 機能は必須ではない  
今回は、命令型としてのシンプルさを求めているため return による jump を導入

## Return による jump 機能の実装②

値を返すのは return だけ →

Null の場合と値が得られた場合でパターンマッチ

→Maybe モナドの恩恵が受けられそう！

## Return による jump 機能の実装③

Sequence の表現:

```
evalStatement env (Seq []) = return Nothing
```

```
evalStatement env (Seq (h:t))
  = do
    car <- evalStatement env h
    maybe(evalStatement env (Seq t))(return . Just)car
```

Nothing で評価続行, Just で評価停止 (→ なんか逆...)

# デモ

```
>> x := 1; return x; x := 10
```

```
1
```

```
>> return x
```

```
1
```

# まとめ

return 文の実装により, 関数の返り値の指定が可能になった

# AST

構文的には, 関数の宣言, 関数適用が必要

```
data Expr = ...  
          | Func String Statement  
          | Apply Expr Expr
```

今回は, どちらも Expr に組み込む



# 実際の構文

Func:

```
func_name := func(arg){body}
```

Apply:

```
var := func_name(param)
```

# Parser

Func の Parser . . . 愚直に実装できる

Apply の Parser . . . 愚直に実装すると...

```
parseApply :: Parser Expr
parseApply = do func <- parseExpr
               param <- parens parseExpr
               return $ Apply func param
```

→予約語等で縛っていないため失敗条件(?)が緩い,  
優先度の高い位置に置くと失敗せずに無限ループに

演算子として見る, 優先度高め → buildExpressionParser に組み込む

# parseApply

専用の演算子, 予約語を使わずに実装したい

```
parseExpr = buildExpressionParser
  [[postfix (parens parseExpr) Apply]
   ,[binary "^" Pow AssocRight]
   .
   .
  ]
  exprTerm
where
  .
  .
  postfix args fun = Postfix (flip fun <$> args)
```

→ (引数) を後置演算子として使用する (微妙?)

# Func の評価

関数型のようなものを用意し, 関数型を返す

イメージ:

```
evalExpr env (Func arg body)
  = return (関数型の値)
```

# 関数型の実装案①

案 1:

```
data TypeEnv = ...  
              | Closure Env String Statement
```

Closure 内の Env の入れ子でスコープを表現する

## 案①による Apply の評価①

バラして, 入れて, 中身进行评估

手順:

- (1) `func_name` を評価 → 関数なら `Closure` が返ってくる
- (2) もし `Closure closureEnv arg body` が返ってきたら...
  - (i) `ClosureEnv` に `(arg param)` を `defineVar`
  - (ii) `evalStatement closureEnv body`

## 案①による Apply の評価②

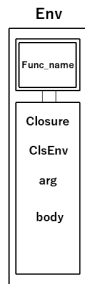
```

evalExpr env (Apply funcname param) = do
  func <- evalExpr env funcname
  case func of
    Closure closureEnv arg body -> do
      value <- evalExpr env param
      (i)      newenv <- defineVar closureEnv arg value
      (ii)     result <- evalStatement closureEnv body
              maybe (return Null) return result
      _ -> error "Error in func"

```

# 関数定義, 関数適用の際の環境の動き①

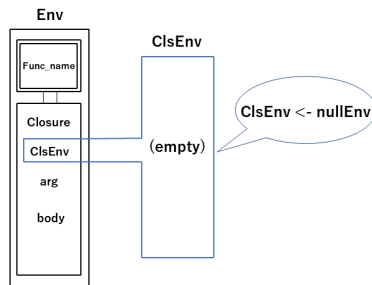
## 1. funcの宣言 `func_name := func(arg){body}`





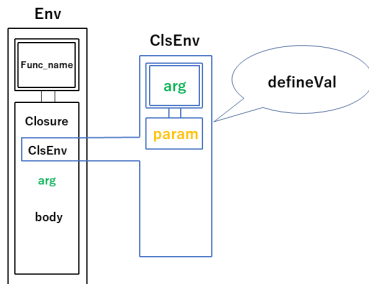
# 関数定義, 関数適用の際の環境の動き②

## 1. funcの宣言 `func_name := func(arg){body}`



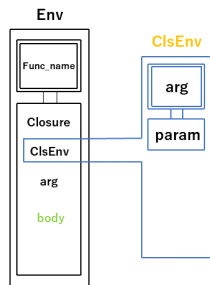
# 関数定義, 関数適用の際の環境の動き③

## 2. Apply func\_name(param)



# 関数定義, 関数適用の際の環境の動き③

## 3. evalStatement CIsEnv body



## 案①の問題点

- 関数内から外側のスコープには一切アクセスできない
- スコープが関数宣言時に確定している

# まとめ

環境を入れ子構造にすることにより関数実行時のスコープを表現したが、閉じた関数しか表現できないという問題点がある

## 再帰呼び出しの構文

(すごい雑に表すと)

```
hoge := func(x){x := x + 1; return hoge(x)}
```

# 現状の問題点

- 環境問題 (案①だとガチガチすぎ)

スコープを取る動作を, 関数宣言時ではなく関数適用時にする必要がある

## (補足) 未定義関数の Apply について

評価の順番を考えると納得  
func 宣言時, body は評価していない



# 環境問題解決の方針

スコープを, ただの連想リストの入れ子で表現していた

→ スタックっぽくする

# 環境問題の解決

内側に新しい空の環境を取っていた

→ もとの環境を参照する形にする.

```
evalExpr env (Func arg body) =  
    return (Closure env arg body)
```

(Closure の中身は案①のまま)

これにより, 関数が開けた

## 案②: 環境をスタック化する

新たな Env の定義:

```
type Env = IORef [Map String (IORef TypeEnv)]
```

Env を, Map の List で定義  
List をスタックとして扱う

## スタックにまつわるヘルパー関数:push

```
push :: Env -> String -> TypeEnv -> IO Env
push env var val = do
    valRef <- newIORef val
    cons <- readIORef env
    newIORef (Data.Map.fromList [(var, valRef)]:cons)
```

push は, List の先頭に新たな要素を追加

# スタックにまつわるヘルパー関数:pop

```
pop :: Env -> IO Env
pop env = do
    garbage <- readIORef env
    case garbage of
        (h:t) -> do newIORef t
        [] -> error "stack error"
```

pop は, List の car を破棄

## その他のヘルパー関数の変更

スタック (List) の頭から順に捜査していく

例: `getVal` の場合

```
getVal :: Env -> String -> IO TypeEnv
getVal envRef var = do
  envStack <- readIORef envRef
  case envStack of
    (h:t) -> do
      cdr <- newIORef t
      maybe (getVal cdr var) readIORef (lookup var h)
    [] -> return Null
```

データ捜査は List としての性質を用いる

# スタックを使用した関数適用の挙動

手順:

- (1) `func_name` を評価→関数なら Closure が返ってくる
- (2) もし Closure `closureEnv arg body` が返ってきたら...
  - (i) ClosureEnv に, `(arg param)` を組み込んだ  
新たな実行コンテキストを push
  - (ii) `evalStatement newEnv body`
  - (iii) 使い終わった実行コンテキストは pop

## 案②による Apply の評価

```
evalExpr env (Apply funcname param) = do
  func <- evalExpr env funcname
  case func of
    Closure closureEnv arg body -> do
      value <- evalExpr env param
      (i)      newenv <- push closureEnv arg value
      (ii)     result <- evalStatement newenv body
      (iii)    garbage <- pop newenv
              maybe (return Null) return result
              _ -> error "Error in func"
```

スコープの挙動はスタックベース



# デモ

```
(>> a := 100)
```

```
>> hoge := func(x){a := 5; return a}
```

```
>> return hoge(1)
```

```
>> return a
```

aが定義される場所によるスコープの挙動の差

## 両ケースに共通する動作

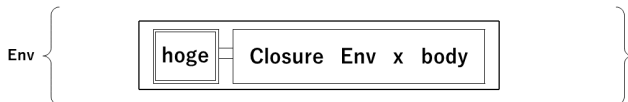
- (i) 引数の(識別子, 値)を, 実行コンテキストをとって push
- (iii) 使い終わった実行コンテキストを pop

→ (ii) の evalStatement だけが異なる, 特に defineVal

[https://github.com/tkyawa/project\\_research\\_a/blob/master/rec/Env.hs](https://github.com/tkyawa/project_research_a/blob/master/rec/Env.hs)

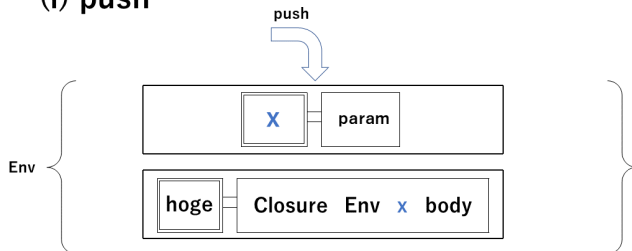
# スキープの挙動① local 変数の場合

## 1. funcの宣言



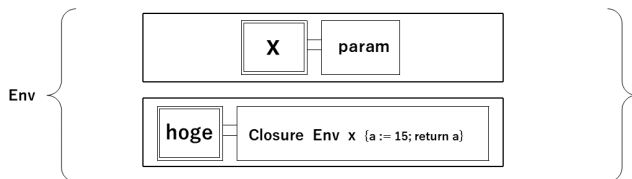
# スキープの挙動① local 変数の場合

## 2. Apply (i) push



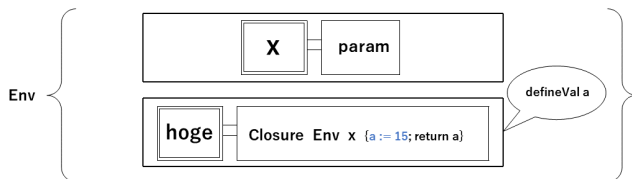
# スキープの挙動① local 変数の場合

## 2. Apply (ii) evalStatement



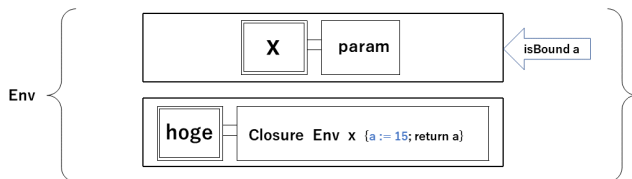
# スキープの挙動① local 変数の場合

## 2. Apply (ii) evalStatement



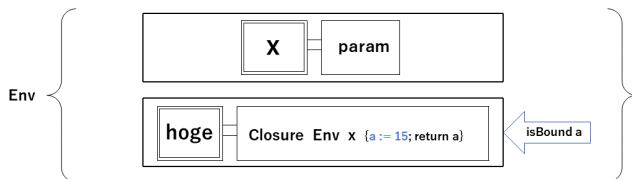
# スキープの挙動① local 変数の場合

## 2. Apply (ii) evalStatement



# スキープの挙動① local 変数の場合

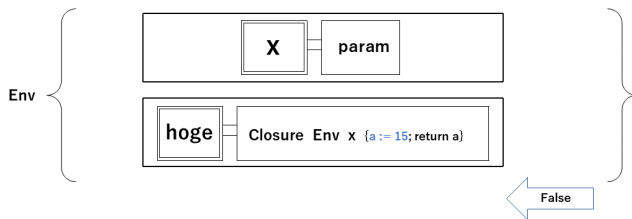
## 2. Apply (ii) evalStatement





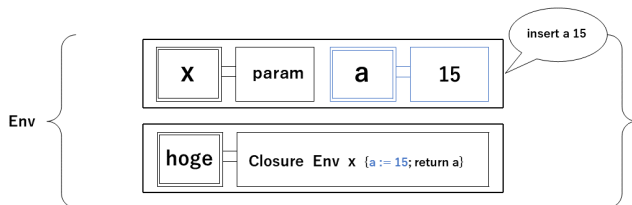
## スコープの挙動① local 変数の場合

## 2. Apply

(ii) `evalStatement`

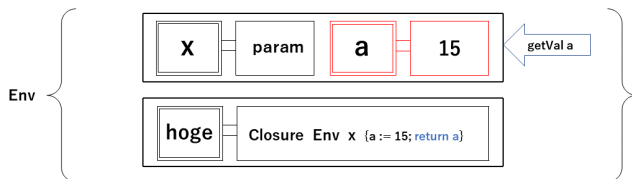
# スキープの挙動① local 変数の場合

## 2. Apply (ii) evalStatement



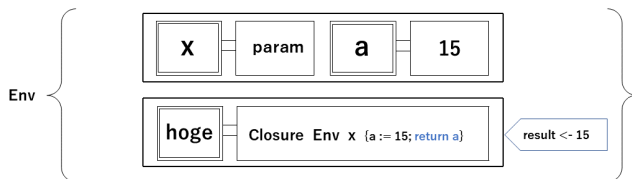
# スキープの挙動① local 変数の場合

## 2. Apply (ii) evalStatement



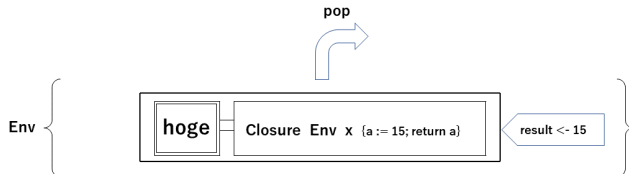
# スキープの挙動① local 変数の場合

## 2. Apply (ii) evalStatement



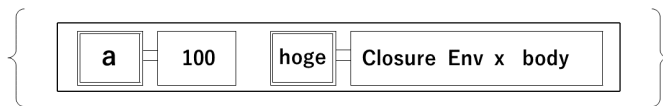
# スコープの挙動① local 変数の場合

## 2. Apply (iii) pop



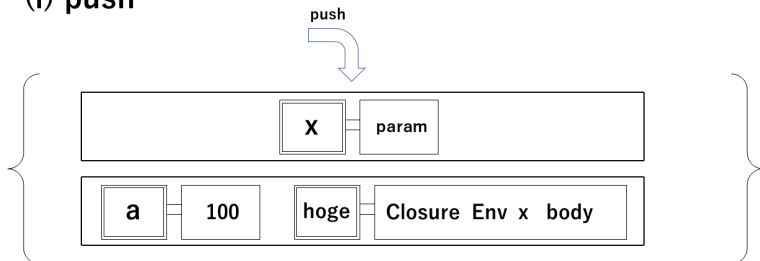
# スキープの挙動② global 変数の場合

## 1. funcの宣言



# スキープの挙動② global 変数の場合

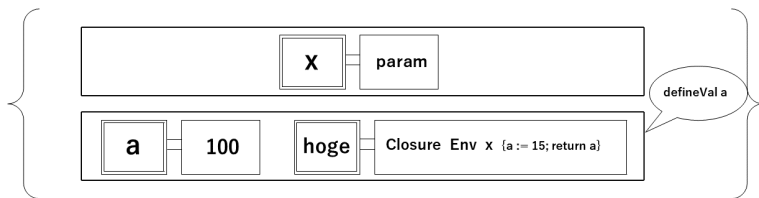
## 2. Apply (i) push



## スコープの挙動② global 変数の場合

## 2. Apply

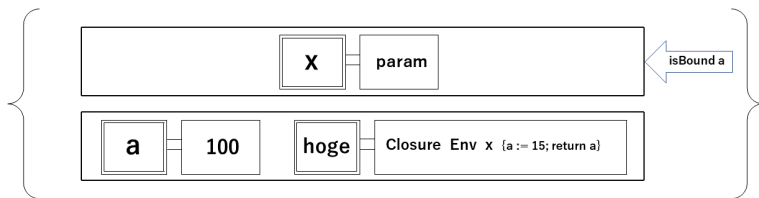
### (ii) evalStatement





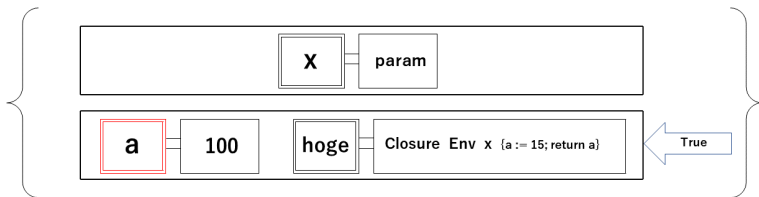
# スキープの挙動② global 変数の場合

## 2. Apply (ii) evalStatement



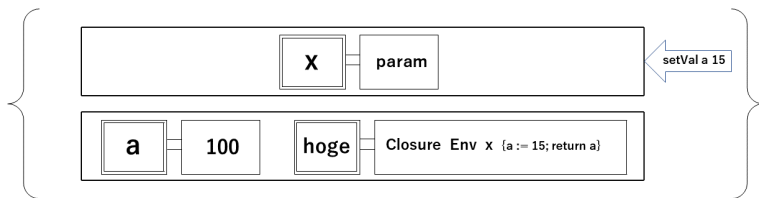
# スキープの挙動② global 変数の場合

## 2. Apply (ii) evalStatement



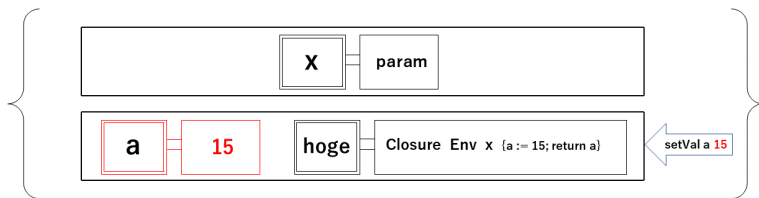
# スキープの挙動② global 変数の場合

## 2. Apply (ii) evalStatement



# スキープの挙動② global 変数の場合

## 2. Apply (ii) evalStatement



# まとめ

環境をスタックで実装したことにより、スコープの表現と柔軟性が両立し、再帰呼び出しが行えるようになった

## 更なる拡張を考える

- 多引数の関数
- 文法の整理
- List の実装
- アクセス修飾子の実装

# 多引数の関数

カーリー化で実現可能

例: add

```
add := func(x){return func(y){return x + y}}
```

実際に構文に組み込む際は, このシュガーとして実装

## 文法の整理

二項演算等をプリミティブな関数として用意すれば, Expr に無理に組み込む必要がなくなる?

もしくは zero, succ のみ用意して組んでいく?



# List の実装

List 型, cons 演算子等をプリミティブに用意すれば, 再帰呼び出しが可能なので List 内の要素へのアクセスも可能

# アクセス修飾子の実装

グローバルなスコープがスタックの底だとわかっているから、それを元に実装可能？

## まとめ, 感想

実装を通して, 言語設計における悩みどころが体感できた

- 何を式, 何を文とするか
- 各シンボルの導入, 構文的な役割 (; や等)
- 関数実装周りのあれこれ (構文的な位置付け, return, 環境, etc...)

## 参考文献

- [1] Thorsten Ball, 設楽 洋爾, Go 言語でつくるインタプリタ, オライリージャパン, 2018 年
- [2] Daniel P.Friedman, Matthias Felleisen, 元吉 文男, 横山 晶一, Scheme 手習い, オーム社, 2010 年
- [3] [https://en.wikibooks.org/wiki/Write\\_Yourself\\_a\\_Scheme\\_in\\_48\\_Hours](https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours)  
最終アクセス: 2021 年 8 月 1 日