関数型プログラミングを特訓する小さな言語

倉光 君郎[1,a)]

概要：
関数型プログラミングを学ぶことはよいプログラミングを学ぶ近道である．しかし，言語設計が（理論的に）おしゃれ過ぎたりすぎたり，マルチパラダイムゆえの複雑さなど，様々な理由から練習しやすい言語は少ない．そこで，我々は関数型プログラミングを練習するためのコンパクト言語として Chibi を設計した．Chibi の設計思想は，使いやすさより練習しやすさに焦点をあてることである．関数型プログラミングを学ぶのに余分な言語機能は排除した．排除された主な機能は，破壊的代入，ループ構造，制御構造，クラス（オブジェクト指向）などである．多少，実用的なプログラミングをするため，Python や JavaScript からデータ構造を得た．これらを組み合わせ，高階関数やパターンマッチング，モナドプログラミングなど，現代的な関数型プログラミングの練習ができる．本稿では，Chibi の試作を用いて，Chibi をチュートリアル的に紹介する．

Keywords: 関数型プログラミング，言語設計，文法設計，プログラミング教育

1. はじめに

近年リリースされるプログラミング言語の特徴は，関数型プログラミング (functional programming) を積極的に採用している点である．Java や C++[7] は，λ式を導入し，高階関数を活用したプログラミングをより書きやすくバージョンアップしている．また，Rust では，λ式だけでなく，パターンマッチや代数的データ構造 (union)，不変データなどのアイディアを取り入れ，新しいシステム記述言語の分野でもより本格的な関数型プログラミングを可能にしている．

これらの動向の背景には，関数型プログラミングが堅牢で高信頼なプログラムを構築する上で，有益であるというコンセンサスがソフトウェア技術者の間に広まったことを示している．学ぶものの立場から見ると，関数型プログラミングを学ぶことははよいプログラミングを学ぶ近道といえる．

しかしながら，関数型プログラミングを練習するよい言語 [8] は少ない．Haskell や OCaml は，関数型プログラミングの発展に大きく貢献してきたが，理論面の主張が強い傾向があり，言語処理系の特徴のみならず，特有の構文につながっている．意外と，構文レベルで苦労することも多い．

一方，マルチパラダイム関数型言語は，オブジェクト指向プログラミング言語などに関数型プログラミングを取り込み，使いやすさ [5] を向上させている．しかし，関数型プログラミングを使わなくてもコーディングが可能 [4] であり，意識して練習しないと関数型プログラミングを学びにくい．

我々は，Chibi (Konoha) と名付けた関数型プロ

[1] 横浜国立大学大学院，Graduate School of Computer Engineering, Yokohama National University, JAPAN
[a)] kimio@ynu.ac.jp

グラミングの練習に特化した小さな言語の設計と開発を進めている．

設計方針は以下のとおりである．まず，我々が開発してきた静的スクリプト言語 Konoha [1], [6] から次の機能を取り除くことで関数型言語に不要な機能を取り除いている．

- 破壊的代入
- ループ構造 (while, for)
- ジャンプ (return, break, continue)
- クラス
- NULL

関数型プログラミングの代表的な機能を追加している．

- 高階関数
- アドホック多相
- 構造的部分型付け
- パターンマッチ

Chibi は，Konoha とは異なる別の言語といえる．本論文では，紙面の都合であまり触れることができないが，新しい実装方法を用いている．構文定義は，純宣言拡張された PEG[2], [3] を用い，Web ページのテンプレートやスキンのようにプログラミング言語の構文を書き換えることができる．また，ORIGAMI トランスパイラフレームワークを用い，複数のソースコードに変換可能なトランスパイラとして開発されている．

本稿は，Chibi の機能と言語的特徴をチュートリアル形式で紹介しながら，最後に言語設計について論じる．

2. Chibi ひとめぐり

Chibi は，ORIGAMI トランスパイラ[*1]に付属しており，`chibi` サブコマンドとして対話シェルが起動できる．

```
$ java -jar origami.jar chibi
Chibi-0.1.446 on Java JVM-1.8.0_144

>>>
```

Chibi の対話シェルは，Python のそれとよく似ている．`>>>` は，対話シェルのプロンプトであり，そこにプログラムを入力すると評価される．

```
>>> 1+2
(Int) 3
```

ちなみに，`(Int)` は評価値が Int 型であることを示している．Chibi は，静的に型付けされた言語である．ユーザは，ほとんど型を意識することなく利用できるが，型を学べばより保守性と信頼性に優れたコーディングを行うことが可能である．

Chibi の文法は，Python よりも，より普遍的な数学記法を多用している．次は関数定義とその関数適用（関数呼び出し）の例である．

```
>>> f(x) = x + 1
>>> f(1)
(Int) 2


>>>
fibo (n) =
  | 1,2 ⇒ 1
  | otherwise ⇒ fibo(n-1) + fibo(n-2)
```

高校数学の知識があれば，初心者でも文法を覚える煩わしさを避けて，直ぐに関数型プログラミングを試すことができる．

3. 構文

Chibi の構文はシンプルである．プログラムは式 (expr) であり，コア構文は次のとおり定義される．

```
expr =
  {expr; expr; ... }
  let name = expr
  if (expr) expr else expr
  expr (expr,...)
  (type) expr
  name
  value
```

Chibi は，ループなどの制御構造をサポートしていない．これは，関数型プログラミングの練習により焦点をあてるためである．一方，Chibi は，優れたパーサ技術により，多くの糖衣構文をもち，

[*1] http://github.com/kkuramitsu/origami

様々な書き方を許容している．

3.1 ブロック

ブロックは，複数の式を順番に評価し，最後の式の評価値を得る式である．最後に評価された値がブロックの評価値となる．

```
{e1; e2; e3}
```

複数の行にわたるブロックを書くときは，インデントベースの記法を用いてもよい．

```
main() =
  println("hello")
  println("world")
```

3.2 Let 束縛

let 束縛は，名前に左式を割り当てる式である．トップレベルで let 束縛すると，グローバル定数となる．関数 (ブロック内) で let 束縛すると，ローカル定数となる．

```
let x = 2
let y = 3
x + y
```

スコープはブロック単位であり，同じ名前を用いると再定義 (shadowing) される．

```
let x = {
  let x = 2
  let y = 3
  x+y
}
x
```

let は省略しても構わない．同じ名前を使うときは，let を明示的に用いた方がよい．

3.3 If 式

if 式は，条件 *cond* が *true* の場合は then 節 (*e1*)，そうでなければ else 節 *e2* を評価する式です．三項演算子 (ternary) に相当する条件式です．

```
if cond then e1 else e2
if (cond) e1 else e2
```

```
let abs(n) =
  if n < 0 then -1 else n
```

else 節は常に必要です．ただし， then 節が空型のときは省略しても構いません．

```
let square(n) =
  if (n < 0) then println(n)
  n * n
```

3.4 関数適用

関数適用は，標準的な数式記法を採用し，引数は ( ) で囲んで指定する．これは，同時にメソッドコールの完全な糖衣構文である．(つまり，どちらで書いても同じである．)

```
f(x, y, z)
x.f(y, z)
```

Chibi は，暗黙的にカリー化しない．むしろ，現在は未サポートである．将来は，次のような明治的なカリー化が予定されており，そのときは平置 (juxtaposition) スタイルの関数適用が可能になる計画である．

```
curry f x y z
```

3.5 型による変換

Chibi の特徴は，型による変換を特別な意味論の関数として扱い，キャストスタイルの記法で扱うことができる．

```
(String)1.0 // conversion
```

Chibi 内部では，変換関数はソースとターゲットの型で識別される．つまり， Float から String への変換関数は， (Float→ String) である．

変換を独立的に扱うメリットは，関数名を覚える煩わしさを軽減することだ．また変換関数の合成も可能になる．

```
(Float → Float) (\n n+1)
```

注 意: ((Int → Int) → (Float → Float)) は，(Float→ Int) と (Int→ Float) から合成可能である．

Chibi は，将来的，変換をより使いやすくするため，合成を制御する機能を提供する予定である．

3.6 Loop

Chibi は名前が示すとおりコンパクト指向であり，ループはない．よく知られていることにループは，再帰 (recursion) で書き換えることができる．ループと再帰を比較すれば，一般に，再帰は関数の性質 (what) を示し，ループの手続き (how) より可読性と保守性が高くなる．

```
gcd(x, y) =
  if y = 0 then x else gcd(y, x - y)
```

しかし，すべてを再帰で書くのは少々無理強いというものである．ループに相当するコードをリストと高階関数で実現できるようにしている．

[0 to <N] は，0 から N 未満の整数の順序リストである．これと *forEach* を組み合わせれば，C 言語の for 文に相当する繰り返しがかける．

```
[0 to <10].forEach(\n {
  println (n+1)
})
```

[← e] は，簡易的なジェネレータであり，e の評価値が None でない限り，繰り返し，そこから得られた値からリストを構築してゆく．

```
[← readline()].forEach(\s {
  println (s)
})
```

Chibi は，少なくとも JVM 実装では，遅延リスト (Stream) を用いているため，必要に応じて処理される．

4. データ型と値

まず，基本的なデータと値をみながら，Chibi の概要を把握してゆこう．

4.1 論理値と数値

Chibi は，次の型で論理値と数値を表す．

- Bool 型- 論理値 (例. *true*, *false*)
- Int 型 - 32 ビット以上の整数 (例. 123, 0*xff*)
- Float 型 - 浮動小数点数 (double 相当) (例. 1.23)

演算子は，C/C++ に由来する標準的な演算子が利用できる．ただし，演算子の優先度は，より数学的に自然な流儀で再定義されている．

1	|| or	または
2	&& and	かつ
3	== != < <= > >=	比較演算
4	→ ←→	ならば
5	:: ++	cons, 結合
6	+ -	加算, 減算
7	/ %	徐算, 余算
8	* ^	乗算, べき算

4.2 文字列

String 型は文字列を表す型である．文字列は，ダブルクオートで囲んで表現する．

```
"hello , world"
```

文字コードのバイト数に関係なく，文字数で長さを数える．

```
>>> |"いろは"|
(Int) 3
```

文字列の連結は，トリプルクオートによる文字列テンプレート (string interpolation) の利用を推奨している．

```
>>> s = "world"
>>> '''hello, ${s}'''
```

4.3 タプル

タプルは，2 つ以上の値を組にしたデータ構造である．異なる種類のデータ値を含めることもできる．

```
>>> (1,2)
(Int*Int) (1,2)
>>> (true, 1, 2)
(Bool*Int*Int) (true,1,2)
```

タプルから値を取り出すときは，let 束縛を用いて変数に束縛して取り出す．不要な値は，_ で無視できる．

```
>>> (a, _, b) = (1, 2, 3)
```

4.4 リスト

リストは，同じ型の複数個からなる値の集まりである．T 型のリストは，List[T] 型となる．

```
>>> [1,2,3]
(List[Int]) [1,2,3]
```

リストは，古典的な配列に似た操作を提供し，要素の数を得て，n 番目要素を参照できる．

```
>>> a = [1,2,3]
>>> |a|
(Int) 3
>>> a[0]
(Int) 1
```

リストは，配列と異なり，変更することができない．関数型データ構造として利用するための cons を備えている．

```
>>> 1 :: a
(List[Int]) [1,1,2,3]
```

リストは，*map*, *filter* , *reduce* などの高階関数を提供している．

4.5 関数 (λ 式)

関数は値である．λ 記号に由来するバックスラッシュ \ に続き，パラメータとパラメータ化された式で定義できる．

```
\() 1
\x x+1
\x \y x+y
```

パラメータの型は型推論により決まるが，型アノテーションを与えることもできる．

```
>>> \n : Int n+1
(Int→ Int) C$3@7a117dc2
```

変数に let 束縛すれば，関数として参照できる．

```
>>> let succ = \n : Int n+1
>>> succ(0)
(Int) 1
```

Chibi のλ式は，いわゆるクロージャである．レキシカルスコープ内の自由変数を参照することができる．

```
let add(n) = \x x + n
```

4.6 空値と存在: Option[T] 型

空値は，関数が何も返さないときに返す重要な値である．

```
()
```

Chibi は，シングルトンな値 (型の値がひとつしか存在しない値) はわざわざ型名と区別しない．つまり，() の型は，そのまま () 型である．

Option[T] は，T 型の値の存在/不在を明示的に示す型である．Some(v) は，T 型の v を Option[T] 型に変換する．もし値が存在しないときは，None を用いる．

```
a = Some(1)
a' = None
```

Option[T] 型は T 型の操作をサポートしない．操作したいときは，T 型の値へアンラップして用いる．強制的なアンラップは，値への変換で行う．

```
(Int!!)a   // Option[Int] ⇒ Int
```

強制的なアンラップは，値が存在しないとき例外となる．通常は，*some*?(a) で a の存在を確認し，Option[T] から T 型へアンラップする．こちらは，スマートキャスト (smart cast) と呼ばれ，自動的にアンラップして，then 節の変数 a にバインドしている．

```
square = if some?(a) Some(a*a) else None
```

Option[T] は要素が 0 か 1 つのリストとみなせる．map 関数を使うと，スマートキャストと同じになる．

```
square = a.map(\n n*n)
```

更に，モナド則を備えた flatMap を備えている．こちらは，ネストされた Option[T] を扱うとき便利である．

```
a = Some(1)
b = Some(2)
sum = a.flatMap (\m b.flatMap (\n Some (m + n)))
```

5. レコードと名前辞書

5.1 レコード

レコードは，オブジェクトや構造体に相当するデータ構造である．Chibi は，宣言なしで構築できる．

```
data ={x: 0.0, y: 0.0, z: 0.0}
```

フィールドの型は，(名前からの）型推論で決まる．だから，宣言なしでも静的に型がきまる．

データは変更可能であり，フィールド値は変更できる．ただし，フィールドを追加することはできない．

```
data.x = data.x + 1.0 // OK
data.w = 1.0 // Error
```

functional programming においては，データよりレコードを使うことは望ましい．変更する必要がなくなったらレコードに変換するのが望ましい．

データとレコードは，構造的部分型付けを採用している．パターンマッチングでは，構造的部分型付けにもとづいて，フィールド値の値の存在が確認される．このとき，データとレコードは区別されない．{ }はデータ型のトップであり，任意のデータの上位型となる．

5.2 名前辞書

Chibi の面白いところは，スクリプト言語の柔軟さと静的言語の厳密さが共存していることだ．それを可能にしているのが名前辞書である．驚くかも知れないが，フィールド名の型はコード全体でユニークに決まることになっている．

名前辞書は，ユーザがトップレベルで assume を使って定義できる．

```
assume
  name: String
  age: Int
  Person : {name, age}
  persons : List[Person]
```

このように定義すると，フィールド name は常に String 型, age は Int 型となる．name と age からなるレコードの型は Person として参照できる．List[Person] のように使うことができる．

さらに，型変数や自己参照型 (_) を用いることで，簡単なジェネリックな再帰型も定義できる．

```
assume
  value: a
  next: Option[_]
  Cell: {value, next}
```

名前辞書の利点は，型が明らかに決まる以外に，型と名前が統一されコードの見通しがよくなることがある．一方，大規模コードのグループ開発では統一性を保つかものが難しいかも知れない．が，心配はいらない．Chibi は小規模の練習向けの言語である．もし大規模なコードを書くこと人が現れたら，将来的には名前空間を用いて対応してゆきたい．

5.3 名前からの型推論

Chibi は，実はもう一歩進んで，名前辞書からパラメータの型も推論できる．この機能を名前からの型推論と呼ぶ．

パラメータの型は，型アノテーションが省略されたら，名前辞書から名前規則 (??) にもとづいて推論される．基本，名前 y が未定義なとき，その後方部分名 x が名前辞書で T 型であれば，名前 y は T 型になる．つまり，firstname は，名前辞書の name の型となる．

名前規則において，英大文字/英小文字，アンダースコアは無視される．変数名 Name, NAME, firstName, first_name も全て String 型になる．

いくつか組み込まれた型ルールも存在する．

Bool	x?, is$x$
List[T]	xList, xs
Dict[T]	xDict

1 文字変数は，あまり推奨される名前ではないが，抽象度の高い関数では便利である．特殊なルールとして，ヒント辞書に掲載のない 1 文字変数に

かぎり，型推論を認めている．

名前からの型推論が適用される場合を述べる．

- フィールド名 - 常に名前からの型推論が優先される．例外はない．
- パラメータ名 - 型アノテーションが優先される．型アノテーションが省略されたとき，名前からの型推論を行う．
- let 変数名 - 型アノテーション，右辺式からの型推論が行われる．(名前からの型推論は用いられない．)

6. Mutable Data

Chibi は，副作用の利用を抑制しながら，適切な範囲で副作用を認める．これが可変データであり，レコード，リスト，辞書に対してのみ，可変版を用意している．

使い方は簡単である．データリテラルの先頭に$をつけると可変版になる．

	Immutable	Mutable
レコード	`{x: 0, y:1}`	`${x: 0, y:1}`
リスト	`[0, 1]`	`$[0, 1]`
辞書	`{"x": 0, "y":1}`	`${"x": 0, "y":1}`

変更可能なリストは，可変長配列である．0 から始まるインデックスで要素を参照し，変更することができる．また，push で最後尾に値を追加し，pop で最後尾の値を取ることができる．

```
>>> a = $[0,1]
>>> a[0] = 2
>>> a
[2,1]
>>> a.push(0)
>>> a
[2,1,0]

let f(n) =
  p = {}
  ...
  p
```

これでデータは，ステートモナドと同じく，局所的な利用に限定し，関数全体でみたら参照等価性を保つことに利用できる．

7. 関数

関数は，プログラムをモジュール化する単位であり，再利用の要である．

7.1 多重定義 Overriding

Chibi は，多重定義が可能である．つまり，同じ名前の関数でもパラメータが異なれば関数定義できる．

```
let max(x: Int, y: Int) = if (x > y) x else y
let max(x: Float, y: Float) = if (x > y) x else y
```

ただし，多重定義はトップレベルに限られる．関数内では，次のとおり，関数定義はλ式の let 束縛に置き換えられる．

```
let max = \x: Int \y: Int if (x > y) x else y
```

Chibi では，演算子は Haskell や ML と同じように関数になっている．(*op*) のように括弧で囲んで識別子とできる．

例えば，もし文字列連結を++を+で行いたいときは，次のように，演算子の多重定義すればよい．

```
let (+) (s: String, s': String) = s ++ s'
```

変換子も (←) として同じように定義できる．

```
let (←) (n: Int) = n != 0
```

7.2 型推論とパラメトリック多相

Chibi は，静的言語であるため，関数のパラメータには型を与える必要がある．しかし，今までの例でも見てきたとおり，パラメータの型は省略できる．省略した場合，名前からの型推論 (5.3) で型が決定される．もし名前から型推論できない場合は型エラー[*2]になる．

Chibi は，一文字変数[*3]だけ例外的に扱うことになっている．一文字変数は，型アノテーションを省略して，名前からの推論も失敗したとき，ML

[*2] 型推論をサポートした言語であっても，ドキュメンテーションのためにパラメータの型は明示化が推奨される．

[*3] 一文字変数は，そもそも望ましくないとされる．しかし，関数やデータ構造を抽象的かつ一時的に扱いたいときは便利である．そこで，一文字変数だけ例外化し，パラメトリック多相に用いている．

型推論にしたがい型推論を行う．Chibi は，アドホック多相や多重定義を認めるため，型が決定できない場合も少なくない．その場合は，型変数によって，パラメータ化される．

```
>>> let f(w) = |w|
>>> f
a→ Int
```

型変数は，関数適用のとき，実引数によって型検査が行われ，多重定義される．（原理的には，C++のテンプレートと同じである．）

7.3 パターンマッチ

パターンマッチは，関数型プログラミングに特徴的に見られる言語機能で，多分岐とデータ抽出を同時に操作できる．パターンマッチといえば正規表現を思い浮かべるかもしれないが，そのデータ構造版と見なしても構わない．

Chibi では，適切なモジュール単位を実現するため，関数定義のときのみ，つまり関数引数に対してのみ，パターンマッチが可能になっている．パターンは， 引数 x に対し，(| $x \Rightarrow y$) のように並記する．

```
let fact(n) =
  | 0 ⇒ 1
  | otherwise ⇒ n * fact (n - 1)
```

２つ以上の引数は，タプルパターンとなる．

```
let and(a,b) =
  | (true, true) ⇒ true
  | (true, false) ⇒ false
  | (false, true) ⇒ false
  | (false, false) ⇒ false
```

引数がリストやレコードの場合は，その内部構造に対してパターンマッチを行い，マッチした値を取り出すことができる．

```
let len(a) =
  | [] ⇒ 0
  | [_, a*] ⇒ 1 + len(a)

show(d) =
  | {text} ⇒ text
  | {left , right} ⇒ '''${left},${right}'''
  | {} ⇒ "unknown"
```

7.4 コードマップ

実際に「 Chibi を試したい」と思ったとき，「ライブラリがあるか」不安に思うだろう．ライブラリを調べる前に，Chibi とその実装を少し理解する必要がある．

Chibi は，純トランスパイラ言語である．最初からソースコード変換によって動作するように設計されており，意味論はコードマップと呼ぶ型付きのソースコード変換規則で定義される．

今まで使ってきた対話シェルは，JVM 版のコードマップで与えられている．例えば，整数演算はコードマップによって，VM 命令や STATIC メソッドコールに変換される．

```
+::(Int,Int)->Int   => 0|iadd
-::(Int,Int)->Int   => 0|isub
*::(Int,Int)->Int   => 0|imul
/::(Int,Int)->Int   => 0|idiv
%::(Int,Int)->Int   => 0|irem
^::(Int,Int)->Int  =>
      S|blue/origami/asm/APIs|pow
```

Chibi は，コードマップを定義することで任意の外部関数を呼び出すことができる．コードマップ先の対象言語を切り替えれば，言語を超えて，ライブラリを活用することができる．

コードマップは，Chibi のソースコードから追加することもできる．

```
codemap JVM
  abs:: Int→ Int ⇒ S:java.lang.Math|abs
```

もし使いたいライブラリがあったら，コードマップを書いて呼び出せるようにすると良い．

8. むすびに

Chibi は，関数型プログラミングを学ぶための小さな言語である．関数型プログラミングを学ぶのに余分な言語機能 (破壊的代入，ループ構造，制御構造，クラス定義) は排除した．多少，実用的な

プログラミングをするため，Python や JavaScript からデータ構造を得た．これらを組み合わせ，高階関数やパターンマッチング，モナドプログラミングなど，現代的な関数型プログラミングの練習ができる．

今後は，より Chibi の言語設計を洗練させて，オープンソース公開を進めてゆきたい．また，Chibi は，言語処理系の開発を学ぶための教材としての活用も考えてゆきたい．

参考文献

[1] Kuramitsu, K.: Konoha: implementing a static scripting language with dynamic behaviors, *S3 '10: Workshop on Self-Sustaining Systems*, New York, NY, USA, ACM, pp. 21–29 (online), DOI: http://doi.acm.org/10.1145/1942793.1942797 (2010).

[2] Kuramitsu, K.: Nez: Practical Open Grammar Language, *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, New York, NY, USA, ACM, pp. 29–42 (online), DOI: 10.1145/2986012.2986019 (2016).

[3] Kuramitsu, K.: A Symbol-Based Extension of Parsing Expression Grammars and Context-Sensitive Packrat Parsing, *Proceedings of ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017 (2017).

[4] Mazinanian, D., Ketkar, A., Tsantalis, N. and Dig, D.: Understanding the Use of Lambda Expressions in Java, *Proc. ACM Program. Lang.*, Vol. 1, No. OOPSLA, pp. 85:1–85:31 (online), DOI: 10.1145/3133909 (2017).

[5] Miller, H., Haller, P., Rytz, L. and Odersky, M.: Functional Programming for All! Scaling a MOOC for Students and Professionals Alike, *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, New York, NY, USA, ACM, pp. 256–263 (online), DOI: 10.1145/2591062.2591161 (2014).

[6] Shida, S., Ide, M. and Kuramitsu, K.: Downsizing Konoha Scripting Langauge Intended for Mindstorms NXT (in Japanese), *IPSJ Transaction on Programming*, Vol. 6, No. 4, pp. 1–9 (2013).

[7] Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J. and Daleiden, P.: An Empirical Study on the Impact of C++ Lambdas and Programmer Experience, *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, New York, NY, USA, ACM, pp. 760–771 (online), DOI: 10.1145/2884781.2884849 (2016).

[8] VanDrunen, T.: The Case for Teaching Functional Programming in Discrete Math, *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, New York, NY, USA, ACM, pp. 81–86 (online), DOI: 10.1145/2048147.2048180 (2011).