プログラミング言語実験 scheme レポート

1810654 谷津直弥

2020 年 8 月 11 日

1　演習 1

1.1　1-1

1.1.1　アルゴリズムに関して

関数型言語の特徴を生かして、再帰で書いた。木がリスト構造で書かれているのでリスト構造のときに car と cdr をそれぞれ再帰することで木のすべての要素をなめることができる。

1.1.2　実行例

```
> (map-tree even? TREE)
(#f (#t (#f #t)) #t (#f #t #f))
> (map-tree (lambda (x) (+ x 2)) TREE)
(3 (4 (5 6)) 8 (9 10 11))
> (map-tree (lambda (x) (* x x)) TREE)
(1 (4 (9 16)) 36 (49 64 81))
> (map-tree (lambda (x) (+ x x)) TREE)
(2 (4 (6 8)) 12 (14 16 18))
> (map-tree (lambda (x) (* (* x x) x)) TREE)
(1 (8 (27 64)) 216 (343 512 729))
```

1.1.3　考察

木をリスト構造で表すことができる点が関数型言語では大きいということがわかった。リストで表現されていると分岐が 2 つで済むため、それぞれに再帰を仕込むだけで基本的に総なめすることが可能であり、とても綺麗にかけると感じた。

1.2　1-2

1.2.1　アルゴリズムに関して

map を使用することで再帰の分岐をなくすことを目標として実装した。map で lambda を関数として持ってきて、引数をして木を渡すことで、その深さのものをすべて lambda 式に通せることを利用した。

1.2.2　実行例

```
> (map-tree2 even? TREE)
```

```
(#f (#t (#f #t)) #t (#f #t #f))
> (map-tree2 (lambda (x) (+ x 2)) TREE)
(3 (4 (5 6)) 8 (9 10 11))
> (map-tree2 (lambda (x) (* x x)) TREE)
(1 (4 (9 16)) 36 (49 64 81))
> (map-tree2 (lambda (x) (+ x x)) TREE)
(2 (4 (6 8)) 12 (14 16 18))
> (map-tree2 (lambda (x) (* (* x x) x)) TREE)
(1 (8 (27 64)) 216 (343 512 729))
```

1.2.3 考察

map の使い方を学んだ。map は基本的にその深さのリストをすべて関数に引数として渡すことができる。そのため、分岐条件がない限りは 1-1 のように書くよりも map を使うだけで全部に探索しますよと表現することが可能であるので、簡潔に書くことができる。

2 演習 2

2.1 2-1

2.1.1 アルゴリズムに関して

基本的に手続き型言語とアルゴリズムは変わらない。再帰するときに深さを-1 して、調べたい深さになったらその要素を返していくように設計した。

2.1.2 実行例

```
> (get-depth alphabet 1)
(b1 b2 b3 b4)
> (get-depth alphabet 3)
(d1 d2 d3 d4 d5 d6)
> (get-depth alphabet 4)
(e1 e2 e3 e4 e5 e6 e7)
> (get-depth tokugawa 1)
(義直 秀忠 頼宣 頼房)
> (get-depth tokugawa 3)
(綱誠 綱吉 綱重 家綱 綱教 頼職 吉宗 綱条 頼候)
> (get-depth tokugawa 5)
(家継 治済 家治 重好 治察 定国 定信)
> (get-depth tokugawa 6)
(家斎 斎敦 斎匡)
```

2.1.3 考察

手続き型言語より簡単に再帰させることができる。map をつかうことで深さを 1 ずつ下げられることが簡潔に記述でき、また return する部分も簡単に賭けるため、非常に簡単に書くことができる。map の関数を指定する部分に様々な関数を書くことで簡潔に書けることが今回の課題ではよく分かった。

2.2 2-2

2.2.1 アルゴリズムに関して

2-1 の関数を使用して作製している。それぞれの深さのリストを get-depth によって求めさせ、その中に探索したいものが含まれているかどうかで調べている。

2.2.2 実行例

```
> (get-cousin alphabet 'b3)
(b1 b2 b3 b4)
> (get-cousin alphabet 'c3)
(c1 c2 c3 c4 c5 c6 c7 c8 c9)
> (get-cousin alphabet 'd1)
(d1 d2 d3 d4 d5 d6)
> (get-cousin alphabet 'e1)
(e1 e2 e3 e4 e5 e6 e7)
> (get-cousin tokugawa '秀忠)
(義直 秀忠 頼宣 頼房)
> (get-cousin tokugawa '吉宗)
(綱誠 綱吉 綱重 家綱 綱教 頼職 吉宗 綱条 頼候)
> (get-cousin tokugawa '家継)
(家継 治済 家治 重好 治察 定国 定信)
> (get-cousin tokugawa '斎匡)
(家斎 斎敦 斎匡)
```

2.2.3 考察

補助関数の使用がこの問題では求められていることだが、よく考えるとこれ以前の課題でも標準搭載の関数を途中でつかっているため、そこまで苦労はしなかった。car や cdr も関数であり、関数型言語では操作はすべて関数という概念で動いている。この問題ではそれを自分で定義したことぐらで、それ以外は既存の動きをしているなと思った。

2.3 2-3

2.3.1 アルゴリズムに関して

まず空のリストを用意し、それを木の下へたどった経路を記録していきながら探索していき、文字列が一致したらその経路を返し、葉であれば空のリストを返す。そしてそれらを myappend によって空のリストを消すようにして実装した。

2.3.2 実行例

```
> (get-path tokugawa '家光)
(家康 秀忠 家光)
> (get-path tokugawa '家治)
(家康 頼宣 光貞 吉宗 家重 家治)
> (get-path tokugawa '家慶)
(家康 頼宣 光貞 吉宗 宗尹 治済 家斎 家慶)
> (get-path tokugawa '斎匡)
(家康 頼宣 光貞 吉宗 宗尹 治済 斎匡)
```

```
> (get-path tokugawa '頼候)
(家康 頼房 頼重 頼候)
```

2.3.3 考察

自由課題なだけあり、難易度が少し上がったなと感じた。下から再帰していくか、上からリストを投げながら再帰していくかでかなり迷ったが、上からリストを投げた方が簡潔に書くことができるなと思い、そのように実装した。このときネックとなった部分が空のリストを投げてしまうことであり、そこは前回作製しているmyappend を再帰で子からリストをもらう際に使うことで、空のリストを削除して親に返すように実装した。様々な関数を使用しているためネストが深くなっており、とても見づらいと感じてしまった。

3 演習 3

3.1 3-1

3.1.1 アルゴリズムに関して

問題の指示通りに実装していくだけである。和差積のときは map によって全部再帰させた。また前回課題でネストが深くなっていったことに嫌気がさしたので、関数を事前に定義して短くまとめ、メイン関数は短くまとめた。

3.1.2 実行例

```
> (diff 'x)
1
> (diff '(+ x 5))
(+ 1 0)
> (diff '(+ (** x 3) (* 2 x) 4))
(+ (* 3 (* 1 (** x 2))) (+ (* 2 1) (* 0 x)) 0)
> (diff '(** (* x 3) 2))
(* 2 (* (+ (* x 0) (* 1 3)) (** (* x 3) 1)))
> (diff '(* (+ x 2) (- (** x 2) x)))
(+ (* (+ x 2) (- (* 2 (* 1 (** x 1))) 1)) (* (+ 1 0) (- (** x 2) x)))
> (diff '(+ (** x 2) (* 4 x) 5))
(+ (* 2 (* 1 (** x 1))) (+ (* 4 1) (* 0 x)) 0)
```

3.1.3 考察

かなり上手く短く書くことができた。事前に関数を定義していくことによって、無駄な () を減らし、簡潔に分岐をかくことができた。前回と同じようにリストを結合する際には myappend を使っているところが美しくないなと感じたが、それ以外はおおむね満足できた。演習三回目にしてようやく関数型言語を理解し始めたと実感できた。関数型言語は設計段階でどれだけ詰めることができるかが勝負だと学んだ。

和差積冪のリストを生成するところは少し冗長であるため 3-3 で変更した。

3.2 3-2

3.2.1 アルゴリズムに関して

3-1 で作った微分のリストに対して文字を代入して評価して、その値をもらい、そこに代入することで傾きと切片をもらい、それらを接続する。

3.2.2 実行例

```
> (tangent '(+ (** x 3) (* -2 (** x 2)) 9) 2)
(+ (* 4 x) 1)
> (tangent '(* (+ x 2) (- (** x 2) x)) 3)
(+ (* 31 x) -63)
> (tangent '(+ (** x 2) (* 4 x) 5) 2)
(+ (* 8 x) 1)
```

3.2.3 考察

let による局所変数によって簡潔に書くことができた。今回の課題のポイントはリストによって記述されている式を scheme はどのように評価するか、だろう。しかしこれは lambda で x という変数に対して代入を行うだけでできる。変数に対して代入をすることができるため、極めて実世界に近い振る舞いを記述することができる。ほかの手続き型言語であればその代入部分を自分で実装する必要があるが、関数型言語ではすべてが同じ要素として扱われており、数字の代入が非常に代入であり、また scheme は eval によって式評価が標準で入っているため、とても簡潔に記述できる。

3.3 3-3

3.3.1 アルゴリズムに関して

すこし長いが、やることはあんまり変わらない。
定数であれば 0
変数であれば文字が一致したら 1 それ以外なら 0
和であれば、cadr と caddr で再帰し、和として接続
差も同様
積であれば、3-1 にようにする
べき乗も同様に 3-1 のようにする。
また和差積冪に関して、リスト作成の部分が助長であると 3-1 を記述した際に感じていたので、そこを関数として定義して簡潔に書くようにした。

3.3.2 実行例

```
> (diff2 'x 'x)
1
> (diff2 'y 'x)
0
> (diff2 '(+ y z) 'z)
(+ 0 1)
```

```
> (diff2 '(* x y) 'x)
(+ (* x 0) (* 1 y))
> (diff2 '(+ (* x x) (* x y) y) 'y)
(+ (+ (* x 0) (* 0 x)) (+ (* x 1) (* 0 y)))
```

3.3.3 考察

かなり簡潔に書くことができた。やることは 3-1 の実数部分と変数部分の比較を少し変えるだけで実装できてしまうが、3-1 の関数が助長で気に入らなかったので、かなり形自体は変えた。やっていることは変わらないが、簡潔にかけているので、よくできたと思っている。どうしても () が増えてしまうと見づらくなってしまうので、そこは注意して書くべきだろう。

3.4 3-4

3.4.1 アルゴリズムに関して

まず 3-1 の関数を使用して冗長なものが含まれているリストをもってきて、それに対して simple 関数を当てていく。リストの先頭文字をみて、和差積冪であれば全体のリストをそれぞれの simple に投げ、その中身の部分は再度 simple で再帰していく。ネストが深くなってしまうのは目に見えているので、そこは切り分けた。

3.4.2 実行例

```
> (simple (diff '(+ x 3)))
1
> (simple (diff '(+ (** x 2) (* 4 x) 5)))
(+ (* 2 x) 4)
> (simple (diff '(* (+ x 2) (- (** x 2) x))))
(+ (* (+ x 2) (- (* 2 x) 1)) (- (** x 2) x))
> (simple (diff '(** (* x 3) 2)))
(* 2 (* 3 (* x 3)))
```

3.4.3 考察

ここまで来てしまえばやることは大したことはなかった。まず比較演算子の後ろ部分は simple で再帰し、最適化する、そして比較演算子を付けなおして、それぞれの simple に結合したものを投げるだけで simple 関数自体は完成する。simple+ などは指示通りかくだけであり、局所変数として設定した non-zero-list を求めてしまえば終わりである。non-zero-list も 0 のとき空のリストを渡して、それを myappend で空のリストを削除するだけで作成することができる。演習が進むにつれ、長い実装を要求されてはいるが、その代わりに実装方針がついているため、思考停止で解けてしまうのはどうなのだろうかと思ってしまった。

4 演習 4

4.0.1 アルゴリズムに関して

scheme の最大の特徴である遅延評価と無限ストリームが活きるように実装する必要がある。エラトステネスのふるいは何度も実装しているが、手続き型言語のように配列を事前に用意して、倍数であればはじくという実装方針で実装すると遅延評価が全く意味をなさない。そのため、素数をストリームに蓄え、 1 要素ごとに

素数かどうかをストリームの全要素で割ることで調べ、素数であればストリームへ追加、素数でなければストリームからはじくというように実装することで、遅延評価と無限ストリームを活かすように実装した。

4.0.2 実行例

```
> (primes 5)
(2 3 5 7 11)
> (primes 10)
(2 3 5 7 11 13 17 19 23 29)
> (primes 100)
(2 ..... 541)   (長いので省略している)
```

4.0.3 考察

無限ストリームと遅延評価、この 2 つが scheme をやる意味なのではないかと感じた。遅延評価によって、必要なときに必要な数だけメモリを割くことができ、無限ストリームで、遅延評価する際に定義を無駄にする必要がなくなる。今回の演習で扱ったエラトステネスのふるいは手続き型言語であれば事前に配列をある程度の大きさで用意するのがベタなやり方だが、関数型言語で今回のようなアルゴリズムであれば、事前に大きさを把握する必要がなく、また最適化が非常に容易である。関数型言語の意義がようやく少し理解できた気がした。

5 演習すべてを通しての感想、考察

すべての問題を解いてみて、やはり関数型言語は手続き型言語のように後から追加していってとりあえず動く、という形に持っていけないのがネックであり、また長所であると感じた。簡潔に書くことがほぼ強制されることと、前提が崩れてしまうとすべておじゃんになるため、慣れないうちは非常に時間がかかる。またわからない人は永遠と理解できないだろうなと感じた。

関数型言語はその名の通り、すべての操作が関数で実行される。そのため事前に関数を定義してしまえばある程度どんなことも関数という 1 つの概念で実装できる。手続き型言語のように様々な手続きを行い、それから関数を実行し、などのことがないため、プログラムが非常に簡潔かつ、どんな操作を行っているかがわかりやすい。1 つの概念のもとですべてを実行できるのは非常に美しいと感じた。

遅延評価と無限ストリームに関して、これこそが scheme をやる意味だと私は感じた。手続き型言語でも遅延評価はセグメントツリーなどで実装したことがあるが、いつ更新するかなどを定義していくことが非常にめんどくさく、わかりづらい挙動をしてしまう。今回の演習ではストリームを生成した際に遅延評価を行うといった、簡単なものではあるが、それでも delay 一つだけで評価されるのはすごく楽で、簡潔に書けるなと関心した。これほど簡単に遅延評価がかけると、関数型言語は最適化に非常に強いと認めざるを得ない。大きなデータを高速で捌くシステムに関数型言語が選ばれる理由が演習 4 まで解いてようやく理解できた。