プログラミング言語実験・Scheme 課題レポート

1510151 柳 裕太

2017 年 8 月 9 日

1 課題 1

1.1 map-tree 関数

1.1.1 設計上の留意点

基本的には、ヒントページ (https://www.ied.inf.uec.ac.jp/text/laboratory/scheme/ex-2017/ex-1-hint1.html) のその 1 の雛形に沿って作成した。ヒントページの空欄には、ドット対でない場合にのみ呼び出される処理が入る。ここで、木の枝 (car，cdr で指定) を対象にもう一度 map-tree 関数を再帰呼出し、結果を cons で繋いでドット対を形成している。

ソースコード 1　map-tree 関数実行例 (対象関数:even?)

```
1  (#f (#t (#f #t)) #t (#f #t #f))
2  ((#f (#t (#f #t)) #t) (#f #t #f))
```

1.1.2 考察

課題の本意通り、map 関数と同等の処理を、S 式によって作成された木構造にも適用することができた。car，cdr で枝を指定することが、処理を実現する最大の決め手になったと考えている。

また、各要素に対する引数に指定された関数の結果を基に、木をまた再構成しているが、こちらも再帰呼出しで結果を問い合わせ、それを基に cons で繋いで引数の木構造を再現できている。これもまた、再帰呼出しを活用するメリットなのではないかと考えている。

1.2 map-tree2 関数

1.2.1 設計上の留意点

基本的には、ヒントページ (https://www.ied.inf.uec.ac.jp/text/laboratory/scheme/ex-2017/ex-1-hint1.html) のその 1 の雛形に沿って作成したものの、空欄部分を変更した。まず map 関数を指定し、使用する関数には lambda 式を使い、「map-tree2 関数に引数として受け取った関数と木の結果を出力する」無名関数を指定し、無名関数への引数としても当該関数が受け取った木を指定している。

ソースコード 2　map-tree2 関数実行例 (対象関数:even?)

```
1  (#f (#t (#f #t)) #t (#f #t #f))
```

```
2   ((#f (#t (#f #t)) #t) (#f #t #f))
```

1.2.2 考察

map-tree 関数では、わざわざ枝を分割して再帰呼出を行っていた。しかしながら、今回の map-tree2 関数では、単純に map 関数で全ての枝の要素を対象に map-tree2 関数再帰呼出を、lambda 式による無名関数を活用して行っている。これは、再帰呼出を行うケースは、受け取った第二引数が部分木である場合に限定されるからであることが考えられる。

2 課題 2

2.1 get-depth 関数

2.1.1 設計上の留意点

家系図の任意の深さまで探索を行い、所定の深さに存在する要素をリスト化するのが目的である。cond で条件わけをする際は、まずデータが空の場合は空を返すように指定した。その次に、第二引数として受け取った depth の値を調べる。もし値が 0 だったならば、木の親を取り出した要素によるリストを返す。それ以外で、もし 1 だった場合は、子の枝を抹消し、親の要素を返す。depth が 2 以上だった場合は get-depth 関数を、子の部分木を第一引数として、第二引数は depth をデクリメントした値を渡す再帰呼び出しを行う。

ただ、そのままだと丸括弧が無駄に残ってしまう (空の部分木が残る) ため、リスト同士を結合する append 関数を、リストの全ての要素を対象とするために apply 関数を活用して 1 次元のシンプルなリストを作成した。

ソースコード 3　get-depth 関数実行例

```
1   (義直 秀忠 頼宣 頼房)
2   (綱誠 綱吉 綱重 家綱 綱教 頼職 吉宗 綱条 頼候)
3   (家斎 斎敦 斎匡)
4   (家康)
```

2.1.2 考察

指定の深さに該当する名前のみの S 式は構成に成功した。これは、depth を再帰呼出しでデクリメントしておく必要性を知ることが可能となったためである。

しかしながら、その S 式を単純なリストにする点で難航した。それは、Scheme がリスト同士を結合して 1 つのリストを構成する apply 関数を、結果として得られた S 式の要素 1 つ 1 つを対象とするために apply 関数を活用する必要がある点に気づくのが遅れた点が挙げられる。

また、今回作成したコードはもう少し改善の余地が挙げられる。特に、以下のコードには改善の余地がある。

ソースコード 4　get-depth 関数一部

```
1   (cond ((null? ls) '() )
2         ((= depth 0) (list (car ls)))
```

```
3           ((= depth 1) (map car (cdr ls)))
4           (else
5             (apply append
6               (map
7                 (lambda (ls)
8                   (get-depth ls (- depth 1))
9                 )
10                (cdr ls)
11              )))
12    )
```

この中で、depth の値が 0、及び 1 だった場合は、それぞれ特別扱いして処理を入れているが、該当部分を 1 行で表現する手段がないか模索したものの、断念に至っている。

2.2 get-cousin 関数

2.2.1 設計上の留意点

この関数では、検索部分と、検索結果の処理部分に分割している。具体的には、実際に第二引数の名前を検索し、該当箇所の深さの数値を返す search 関数と、その結果を基に同じ深さの名前のリストを作成して返す get-cousin 関数となっている。

search 関数では、引数として木と名前と調べる深さの 3 種をもち、もし要素が空であれば 0 を返し、リストの要素内に名前があれば、深さの値を返す。要素はあれど名前が発見されなければ、深さ depth をインクリメントして search 関数を各枝を対象に再帰呼出を行い、帰ってきた数値の総計を返すようになっている。

get-cousin 関数では、まず親を抜いた枝の全要素に search 関数による探索をかけ、それぞれの結果を加算して、その結果で得られた値を get-depth 関数への引数 depth として活用し、戻ってきた結果をそのまま返している。

ソースコード 5　get-cousin 関数実行例

```
1   (義直 秀忠 頼宣 頼房)
2   (綱誠 綱吉 綱重 家綱 綱教 頼職 吉宗 綱条 頼候)
3   (家斎 斎敦 斎匡)
4   (家康)
```

2.2.2 考察

search 関数でも、get-cousin 関数でも lambda 式を使用しているが、無名関数として行っている処理はほぼ同一である (ソースコード参照)。唯一の違いは search 関数を呼び出す際の第二引数が、search 関数ならば depth をインクリメントした数であり、get-cousin 関数ならば最初の枝を表す 1 となっている点となっている。これは、2 関数間であっても、再帰呼出の戦略はほぼ同じであることを示唆している。

3 課題 3

3.1 diff 関数

3.1.1 設計上の留意点

引数として受け取った要素を、まずは cond で場合分けしている。場合分けの順序と方針は、課題 3-1 のヒント (https://www.ied.inf.uec.ac.jp/text/laboratory/scheme/ex-2017/ex-3-1.html) に準拠している。もしもどれにも該当しないならば、子の要素に対し diff 関数を再帰的に適用している。ここでは、加算・減算・乗算・累乗の詳細な処理内容について記載する。

加算/除算 (先頭要素が"+"/"-") であれば、二番目以下の要素全てを対象に diff 関数を再帰的に適用し、帰ってきた値同士を map で加算/除算を行っている。

乗算 (先頭要素が"*") であれば、(+ (* 第 1 引数 第 2 引数の微分) (* 第 1 引数の微分 第 2 引数)) の木を、第 1 引数は (car (cdr ls)) で、第二引数なら (car (cddr ls)) で抜き出し、微分するならこの式を diff の引数として丸括弧で囲うことで対処している。なお、行列の要素なのか計算するべき式なのか区別させるため、準クォートを採用して引数を抜き出す式の前にカンマを置くことで対処している。

累乗 (先頭要素が"**") であれば、乗算と同じ手法で、(* 第 2 引数 (* 第 1 引数の微分 (** 第 1 引数 (- 第 2 引数 1)))) の木を形成している。

ソースコード 6 diff 関数実行例

```
1  1
2  (+ 1 0)
3  (+ (* 2 (* 1 (** x 1))) (- (+ (* 4 1) (* 0 x)) 0))
4  (+ (* 6 (* 1 (** x 5))) (+ (* -2 (* 2 (* 1 (** x 1)))) (* 0 (** x
       2))) 0)
```

3.1.2 考察

準クォートを多用した。特に、乗算・累乗の場合の式の返し方にて、準クォートを使うのはわかったものの、どのタイミングでカンマを入れるべきなのかがこの課題の肝であると読み取れた。何故ならば、カンマの入れるタイミングを誤ると、評価されるべき式 (累乗の"(- 第 2 引数 1)" など) が評価されず、あるいはシンボルを含む評価できない式を評価しようとしてエラーを吐くこととなったためである。

3.2 tangent 関数

3.2.1 設計上の留意点

接線の方程式を求める公式を、Scheme 上で実現する形をとっている。具体的には、1 次の x の項を求める tangent-fa 関数と、定数項を求める tangent-a 関数を作成した。tangent 関数では、第一引数の x の座標 a と第二引数の y 座標 fa を使用して、tangent-fa 関数と tangent-a 関数の返した値をもとに接線の方程式を返すようにしている。

ソースコード 7　tangent 関数実行例 ()

```
1  (+ (* 4 x) 1)
2  (+ (* 184 x) -309)
```

3.2.2 考察

tangent-fa, tangent-a の 2 つの関数の共通点は、eval 式を使用して、a を基に評価した値を返していることが挙げられる。双方の違いは、tangent-fa では微分した式を、tangent-a は式そのものを評価している点である。

また、tangent 関数にて接線の方程式を作る際は、let を用いた。これによって 1 次の x の項を fda、定数項を fa_val に束縛することで、実際に項を得る部分と式を形成する部分を分離して可読性を上げることに成功した。ケースによっては準クォートではなく、let で評価する部分と記述する部分を完全に分離した方が可読性が上がるのではないかと考えている。

3.3 diff2 関数

3.3.1 設計上の留意点

基本的な考え方は diff 関数と大差はない。大きな違いとして挙げられるのは、第二引数として微分する対象の記号を指定することで、シンボルが第二引数のシンボルと全く同じなら 1 を返すようにしたことと、再帰呼び出しする際にも第二引数を追加した点にとどまっている。

ソースコード 8　diff2 関数実行例 ()

```
1  1
2  0
3  (+ (* x 0) (* 1 y))
4  (+ (+ (* y 0) (* 1 x)) (- (* 2 (* 1 (** y 1))) 0))
```

3.3.2 考察

単純な微分から偏微分への改変には、個人的には少々大掛かりな変更が入るのではないかと考えていたが、かなりシンプルな変更で済むこととなった。これもまた、再帰呼出で処理を行う Scheme の長所として考えることができるのではないかと考えた。