プログラミング言語実験 Scheme

1710730 須藤敬仁

2019 年 8 月 16 日

1　課題 1

この課題の実行例では以下の木構造を使用する.
(define TREE '(1 (2 (3 4)) 6 (7 8 9)))
(define MYTREE '(1 2 (3) (4 5 6 (7 8) (9))))

1.1　その 1

木構造全体をある関数によって写像する関数 map-tree を作成する. 関数 map-tree は引数に適用したい関数と写像元となる木構造を入力する. 入力された木の全ての要素に対して関数を適用するため, 再帰的に適用するアトムを辿り, アトムに辿りついたら関数を適用すればよい. アトムへの辿り方は与えられた木の car 部と cdr 部それぞれに対して関数 map-tree を適用してそれを cons で繋げれば良い. これらの操作を与えられた写像元が空リストかドット対かアトムかで条件判別し適用すればよい.

1.1.1　実行例

(map-tree even? TREE) → (#f (#t (#f #t)) #t (#f #t #f))
(map-tree (lambda (x) (* x x)) TREE) → (1 (4 (9 16)) 36 (49 64 81))
(map-tree (lambda (x) (+ x 1)) MYTREE) → (2 3 (4) (5 6 7 (8 9) (10)))

1.2　その 2

その 1 で作成した関数 map-tree を cons, car, cdr を利用せず, map だけで実現する関数 map-tree2 を作成する. map 関数で関数 map-tree2 を再帰的に適用させていけばよい. そうすると, 木構造の階層ごとに関数 map-tree2 が適用されていくこととなる. なので, 関数 map-tree2 は与えられたリストがドット対 (つまり, 部分木) なら再帰的に関数 map-tree2 を適用し, そうでないなら指定した関数をアトムに適用するといった処理内容にすればよい.

1.2.1　実行例

(map-tree2 even? TREE) → (#f (#t (#f #t)) #t (#f #t #f))
(map-tree2 (lambda (x) (* x x)) TREE) → (1 (4 (9 16)) 36 (49 64 81))
(map-tree2 (lambda (x) (> x 4)) MYTREE) → (#f #f (#f) (#f #t #t (#t #t) (#t)))

2 課題 2

2.1 その 1

関数 get-depth には対象の木構造 (この課題では家系図) と深さの 2 つの引数を入力する．木構造の全ての部分木を map によって再帰的に辿り, 目的の深さまで到達したらそのときの木の根 (car 部) 全てについて append を適用すればよい．このとき, list によって木をリストに内包させると深さ 0 の要求にも対応できる．以上の処理を与えられた深さを再帰するごとに-1 していき, 0 になったら条件分岐で木の根ノードを取得するようにすればよい．

2.1.1 実行例

(get-depth kakeizu 0) → (家康)
(get-depth kakeizu 4) → (家宣 宗尹 家重 宗武)
(get-depth kakeizu 6) → (家斎 斎敦 斎匡)

2.2 その 2

get-cousin は指定された名前と同じ深さにある名前の一覧を取得する関数だが, これを実現するために補助関数 search を作成するとよい．この関数は指定された名前の深さを求める関数である．補助関数 search は家系図を再帰的に辿っていき, 終端 (つまり cdr 部が空リスト) に辿りついたら 0, 終端に辿り着く前に指定された名前が存在したらそのときの深さを返すようにして, これらの結果の総和を取れば結果が求められる．そして, get-cousin は関数 search と課題 2 その 1 で作成した get-depth を利用すれば実現できる．

2.2.1 実行例

(search kakeizu ' 治察 0)　→ 5
(get-cousin kakeizu ' 吉宗) → (綱誠 綱吉 綱重 家綱 綱教 頼職 吉宗 綱条 頼候)
(get-cousin kakeizu ' 斎匡) → (家斎 斎敦 斎匡)

2.3 その 3

指定した人物までの経路を求める関数 get-path を作成する．再帰的に関数を呼び出して家系図を階層ごとに見ていくが, それぞれの部分木について根が対象の名前ならばその名前をリストに内包したものを返す．これによってこの関数は処理をリストのみにだけ考えれば良くなる．また, それぞれの部分木について対象の名前が見つからなければその時点で空リストを返す．この機能には課題 2 その 2 で作成した補助関数 search を使用すればよい．最後にこれらを append したものに対して再帰的に呼び出したときの根の名前を cons で先頭に追加していけば完成となる．

2.3.1 実行例

(get-path kakeizu ' 家光) → (家康 秀忠 家光)
(get-path kakeizu ' 家定) → (家康 頼宣 光貞 吉宗 宗尹 治済 家斎 家慶 家定)

3 課題 3

3.1 その 1

与えられた多項式の導関数を求める関数 diff を実装する．与えられた式が乗算または除算のときは第 1 引数と第 2 引数を let で束縛しておくと記述しやすい．また, この関数は x と数値にしか対応していないため例えば'a を入力されたとしても結果は返さない.

3.1.1 実行例

(diff '(+ x 5)) → (+ 1 0)
(diff '(* (+ x 2) (- (** x 2) x))) → (+ (* (+ x 2) (- (* 2 (* 1 (** x 1))) 1)) (* (+ 1 0) (- (** x 2) x)))

3.1.2 考察

x や数値以外のアルファベット等で表された定数に対応するには数値か x か判定しているところにそのような定数を判別するような条件式を加えれば良いと思う.

3.2 その 2

導関数に x を代入した値とその原始関数に x を代入した値は eval によって計算ができ, それぞれ a と b とおく．接線の傾きは a, 切片は b-a*x で求められる．実装する関数 tangent が返すのは方程式なので, シンボルとしての x と変数としての x の区別に注意すること.

3.2.1 実行例

(tangent '(+ (** x 3) (* -2 (** x 2)) 9) 2) → (+ (* 4 x) 1)
(tangent '(+ (** x 4) (* -3 (** x 3)) (* 5 x) 12) -1) → (+ (* -8 x) 3)

3.3 その 3

定数か数値か微分対象かの条件判断をし, 微分対象ならば 1, それ以外ならば 0 とすればよい．そのほかの実装は diff2 に与える引数に気をつければほとんど課題 3 その 1 で実装した関数 diff と変わらない.

3.3.1 実行例

(diff2 '(* x y) 'x) → (+ (* x 0) (* 1 y))
(diff2 '(+ (* a x) (** a 3))'x) → (+ (+ (* a 1) (* 0 x)) (* 3 (* 0 (** a 2))))

3.4 その 4

課題のページにも書いてある通り, まずリストから 0 の項を取り除く関数を実装する．対象が 0 ならば空リスト, それ以外ならリストに内包して返す関数をラムダ式で記述し, それを対象のリストに map する．それをすべて append すれば 0 の項を取り除いたリスト non-zero-list が作成できる．これを利用して関数郡 simple+, simple-を作成する．このとき, let で non-zero-list を変数束縛すると記述が簡潔になる．simple*,

simple**については引数 2 つを let で束縛し, それらの値によって条件分岐して結果を返せば良い. 条件については課題のページで説明されているのでここでは割愛する.

3.4.1 実行例

(simple (diff '(+ (** x 2) (* 4 x) 5))) → (+ (* 2 x) 4)
(simple (tangent '(** x 2) 2)) → (+ (* 4 x) -4)

4 課題 4

まず, 2 以上の無限の整数リスト numbers を作成する. これは遅延評価を利用して実装する. stream-cons によってこれを実現することができる. 次に, 作成したリスト numbers をエラトステネスのふるいにかける関数 sieve を実装する. 関数 sieve には除数と対象のリストを入力する. 対象のリストの先頭を与えた除数で割り, 0 ならばリストの後続に関数 sieve を適用したリストを返す. 0 でないならば stream-cons でリストの先頭とリストの後続に関数 sieve を適用したリストを繋げたリストを返す. また, リスト numbers は遅延評価を用いて実現されているため, 後続を取得するには stream-cdr で呼び出す必要がある. 関数 sieve ではリストの先頭を let で束縛すると記述が楽だが, リストの後続は束縛することはできないので 2 回書く必要がある. これら numbers と sieve を利用して 2 から始まる無限の素数リストを実現する. 実現方法としてはまず numbers の先頭を取得し, stream-cons でこの先頭とその値で numbers をふるいにかけたリストをつなげる. そしてこのふるいにかけた numbers を新たな numbers として再帰的に処理する. このとき, numbers の先頭要素を let で束縛すると記述が楽になる. こうして 2 から始まる無限の素数リストが実現できる. あとは無限リストから stream-cdr で要素を取り出していけば良い. この機能は関数 head として実装している.

4.0.1 実行例

(head 10 (primes)) → (2 3 5 7 11 13 17 19 23 29)