华中科技大学

2019

函数式编程 ·实验报告·

专　　业：	计算机科学与技术
班　　级：	ACM1701
学　　号：	U201714780
姓　　名：	刘晨彦
完成日期：	2020-04-07

计算机科学与技术学院

目　录

1 Lab 1

1.1 代码实现

1.1.1 完成函数 mult 的编写，实现求解整数列表中所有整数的乘积

仿照 sum 函数，mult 函数对传入参数进行匹配，若传入为空串，则返回 1，若传入非空串，则将非空串首元素和剩余元素之积相乘。剩余元素之积通过 mult 函数递归实现。

具体实现代码如下图所示：

```
(* mult : int list -> int     *)
(* REQUIRES: true   *)
(* ENSURES: mult(L) evaluates to the product of the integers in L. *)
fun mult [] = 1
  | mult (x::L) = x * mult(L);
```

图 1.1 mult 函数实现

1.1.2 完成如下函数 Mult: int list list -> int 的编写,该函数调用 mult 实现 int list list 中所有整数乘积的求解

根据题目要求可知，传入 Mult 函数的参数是一个串的集合，需要对两种情况进行匹配：当传入为空串时，则返回 1，若传入为非空串时，串首元素作为一个串，可以调用 mult 函数得到串元素的乘积，并乘以剩下串集中所有串元素的乘积，剩余串中所有串元素的乘积可以通过递归调用 Mult 函数得到。

具体实现代码如下图所示：

```
(* Mult : int list list -> int  *)
(* REQUIRES: true   *)
(* ENSURES: mult(R) evaluates to the product of all the integers in the lists of R. *)
fun Mult [] = 1
  | Mult (r::R) = mult(r) * Mult(R);
```

图 1.2 Mult 函数实现

1.1.3 函数 mult'定义如下，试补充其函数说明，指出该函数的功能。利用 mult'定义函数 Mult'，使对任意整数列表的列表 R 和整数 a，该函数用于计算 a 与列表 R 中所有整数的乘积

已知 mult'的定义如下所示：

fun mult' ([ ], a) = a

| mult' (x :: L, a) = mult' (L, x * a);

根据函数可知，mult'将串 L 的每个元素都和 a 相乘，当 L 为空串时得到 a。故函数的功能是得到串 L 内每个元素和 a 的积。补充函数说明如下图所示：

```
(* mult' : int list * int -> int *)
(* REQUIRES: true        *)
(* ENSURES: mult'(L, a) = mult L * a *)
fun mult' ([], a) = a
  | mult' (x::L, a) = mult'(L, x*a);
(*mult' 得到a和L内元素的乘积*)
```

图 1.3 mult' 的函数说明和功能

要实现 Mult'，则可以利用 Mult 函数来计算。实现如下图所示：

```
(* Mult' : int list list * int -> int *)
(* REQUIRES: true        *)
(* ENSURES: Mult'(L, a) = Mult L * a *)
fun Mult' ([], a) = a
  | Mult' (x::L, a) = Mult'(L, mult'(x, a));
```

图 1.4 Mult' 函数的实现

1.1.4 编写递归函数 square 实现整数平方的计算，要求：程序中可调用函数 double，但不能使用整数乘法（*）运算

根据题目可知，不能使用乘法运算，因此需要利用 double 来实现。根据等式$x^2 = (x-1)^2 + 2(x-1) + 1$，即可利用递归实现。代码实现如下图所示：

```
(* square : int -> int *)
(* REQUIRES: n >= 0 *)
(* ENSURES: double n evaluates to 2 * n.*)
fun square (0: int): int = 0
  | square n = square(n - 1) + double(n - 1) + 1;
```

图 1.5 square 函数的实现

1.1.5 定义函数 divisibleByThree: int -> bool，以使当 n 为 3 的倍数时，divisibleByThree n 为 true，否则为 false。注意：程序中不能使用取余函数’mod’

根据要求，可以利用递归，每次对传入的 n 减去 3，直到$n < 3$，仅当 n=0 时得到 true，其余情况得到 false。函数实现如下图所示：

```
(* divisibleByThree : int -> bool   *)
(* REQUIRES: true       *)
(* ENSURES: divisibleByThree n evaluates to true if n is a multiple of 3 and to false otherwise *)
fun divisibleByThree (0: int): bool = true
  | divisibleByThree (1) = false
  | divisibleByThree (2) = false
  | divisibleByThree (n) = divisibleByThree(n-3);
```

图 1.6 divisibleByThree 函数的实现

1.1.6 编写奇数判断函数 oddP: int -> bool，当且仅当该数为奇数时返回 true

函数实现方法和 divisibleByThree 相似，通过对大于等于 2 的 n 减去 2 并递归，直到 n 小于 2，并判断是否为 0，为 0 则为偶数。具体实现如下图所示：

```
(* oddP : int -> bool      *)
(* REQUIRES: n >= 0      *)
(* ENSURES: oddP n evaluates to true iff n is odd. *)
fun oddP (0: int): bool = false
  | oddP (1) = true
  | oddP (n) = oddP(n-2);
```

图 1.7 oddP 函数的实现

1.2 代码测试

编写测试代码如下图所示：

```
54  (* ----------For tests------------ *)
55  val test_mult =
56      (mult [1,2,3,4] = 24) andalso
57      (mult [] = 1) andalso
58      (mult [5,2,1,8] = 80)
59  val test_Mult =
60      (Mult [[]] = 1) andalso
61      (Mult [] = 1) andalso
62      (Mult [[1,2],[3,4]] = 24)
63  val test_Mult' =
64      (Mult' ([[]], 5) = 5) andalso
65      (Mult' ([], 5) = 5) andalso
66      (Mult' ([[1,2],[3,4]],5) = 120)
67  val test_square =
68      (square 0 = 0) andalso
69      (square 7 = 49)
70  val test_divisibleByThree =
71      (divisibleByThree 3 = true) andalso
72      (divisibleByThree 28 = false) andalso
73      (divisibleByThree 0 = true)
74  val test_oddP =
75      (oddP 81 = true) andalso
76      (oddP 0 = false) andalso
77      (oddP 52 = false)
```

图 1.8 Lab1 测试代码

测试结果如下图所示：

```
- use "D:\\Functional Programming\\Lab1\\Lab1.sml";
[opening D:\Functional Programming\Lab1\Lab1.sml]
val sum = fn : int list -> int
val mult = fn : int list -> int
val Mult = fn : int list list -> int
val mult' = fn : int list * int -> int
val Mult' = fn : int list list * int -> int
val double = fn : int -> int
val square = fn : int -> int
val divisibleByThree = fn : int -> bool
val oddP = fn : int -> bool
val test_mult = true : bool
val test_Mult = true : bool
val test_Mult' = true : bool
val test_square = true : bool
val test_divisibleByThree = true : bool
val test_oddP = true : bool
val it = () : unit
-
```

图 1.9 Lab1 测试情况

根据测试结果可知，代码功能正确。

2 Lab 2

2.1 代码实现

2.1.1 编写函数 reverse 和 reverse’，要求：函数类型均为：int list->int list，功能均为实现输出表参数的逆序输出；函数 reverse 不能借助任何帮助函数；函数 reverse’可以借助帮助函数，时间复杂度为 *O*(n)

对 reverse 函数，若传入为空串，则得到空串，若传入非空串，则将首元素接在剩余串逆序后的尾部即可，代码实现如下图：

```
(* reverse: int list ->int list
   REQUIRES: elements in list are all integer
   ENSURES: reverse a list's order *)
fun reverse[] = []
  | reverse(x::L): int list = (reverse(L)) @ [x];
```

图 2.1 reverse 函数实现

对 reverse’函数，则可以通过辅助函数，每次将输入串 L 中的首元素压入新串 R，最后得到 R。具体实现如下图所示：

```
(* reverse': int list ->int list
   REQUIRES: elements in list are all int
   ENSURES: reverse a list's order *)
fun reverse'(L : int list): int list =
    let fun helpfun(L : int list, R : int list): int list =
        case L of [] => R
                | (x::L') => helpfun(L', x::R)
        in
            helpfun(L, [])
        end;
```

图 2.2 reverse’ 函数实现

2.1.2 编写函数 interleave: int list * int list -> int list，该函数能实现两个 int list 数据的合并，且两个 list 中的元素在结果中交替出现，直至其中一个 int list 数据结束，而另一个 int list 数据中的剩余元素则直接附加至结果数据的尾部

根据题目要求，匹配有四种情况：两串均为空串，则直接返回组装好的新串；若

其中一个串为空串，一个为非空串，则将非空串接在新串后面。若两串均为非空串，则将两非空串的首元素取出并接在新串尾部。新串初始化为空串。具体实现如下图所示：

```
(* interleave: int list * int list -> int list
   REQUIRES: elements in any list are all integer
   ENSURES: merge two list into one*)
fun interleave(L: int list, R: int list):int list =
    let fun helpfun(L: int list, R: int list, Res: int list): int list =
        case (L,R) of ([], []) => Res
                    | (x::L', []) => Res @ (x::L')
                    | ([], y::R') => Res @ (y::R')
                    | (x::L', y::R') => helpfun(L', R', Res@ [x]@ [y])
        in
            helpfun(L, R, [])
        end;
```

图 2.3 interleave 函数的实现

2.1.3 编写函数 listToTree: int list -> tree，将一个表转换成一棵平衡树

函数的实现方式是，将串根据长度一分为二，将中间的元素作为当前树根，并对左右两串递归操作，得到左右子树。具体实现如下图所示：

```
(* listToTree: int list -> int tree
   REQUIRES: elements in list are all intergers
   ENSURES: transform a list into a balanced tree*)
fun listToTree(L: int list): int tree =
    case L
       of [] => Empty
       | _ =>  let
                   val mid:int = length(L) div 2
                   val leftlist = List.take(L,mid)
                   val x:int = hd(List.drop(L,mid))
                   val rightlist = tl(List.drop(L,mid))
               in
                   Node(listToTree(leftlist), x, listToTree(rightlist))
               end;
```

图 2.4 listToTree 函数的实现

2.1.4 编写函数 revT: tree -> tree，对树进行反转，使 trav(revT t) = reverse(trav t)。假设输入参数为一棵平衡二叉树，验证程序的正确性，并分析该函数的执行性能（work 和 span）

对于输入的平衡二叉树，对每个 Node，交换左右子树位置，并对左右子树重复该操作。代码具体实现如下图：

```
(* revT: int list -> int tree
   REQUIRES: elements in list are all intergers
   ENSURES: exchange left-subtree and right-subtree for ever node*)
fun revT(Empty:int tree): int tree = Empty
  | revT(Node(L,x,R)) = Node(revT(R),x,revT(L));
```

图 2.5 revT 函数的实现

假设树的节点数（各子树根节点与叶子节点之和）为 n，则平衡树树高为$\log_2 n$，认为在每一层的交换位置都是并行的，故 span 为：$O(\log n)$。每一层 i 中，有 work：$T(i) = 2^i + T(i+1)$，故总的 work 为：$O(n)$。

2.1.5 编写函数 binarySearch: tree * int -> bool。当输入参数 1 为有序树时，如果树中包含值为参数 2 的节点，则返回 true；否则返回 false。要求：程序中请使用函数 Int.compare（系统提供），不要使用<, =, >

对输入的树，当为空节点时，得到 false，当不为空时，比较当前根节点和参数 2 的大小，若根节点较小，则在右子树中递归，若相同，则返回 true，若更大，则在左子树中递归寻找。代码实现如下：

```
(* binarySearch: int tree * int -> bool
   REQUIRES: elements in tree are ordered
   ENSURES: find x in tree or not*)
fun binarySearch(Empty:int tree, x: int): bool = false
  | binarySearch(Node(L, m, R), x:int) =
    case Int.compare(m,x)
    of GREATER => binarySearch(L,x)
    |  EQUAL => true
    |  LESS => binarySearch(R,x);
```

图 2.6 binarySearch 函数的实现

2.2 代码测试

编写测试程序如下图所示：

```
(* ----Next are for test--- *)
val t:int tree = Node(Node(Node(Empty,1,Empty),2,Node(Empty,3,Empty)),4, Node(Node(Empty,5,Empty),6,Node(Empty,7,Empty)));
val reverse_test = ([1,2,3,4,5,6,7,8,9] = reverse([9,8,7,6,5,4,3,2,1]));
val reverse'_test = ([1,2,3,4,5,6,7,8,9] = reverse'([9,8,7,6,5,4,3,2,1]));
val interleave_test =
    (interleave([2],[4]) = [2,4]) andalso
    (interleave([2,3],[4,5]) = [2,4,3,5]) andalso
    (interleave([2,3],[4,5,6,7,8,9]) = [2,4,3,5,6,7,8,9]) andalso
    (interleave([2,3],[ ]) = [2,3]);
val listToTree_test =
    (listToTree([]) = Empty) andalso
    (listToTree([1,2]) = Node (Node (Empty,1,Empty),2,Empty)) andalso
    (listToTree([1,2,3,4,5,6,7]) = t);
val revT_test = (trav(revT t) = reverse(trav t));
val binarySearch_test =
    (binarySearch(t, 0) = false) andalso
    (binarySearch(t,7) = true) andalso
    (binarySearch(Empty, 4) = false)
```

图 2.7 Lab2 测试代码

```
- use "D:\\Functional Programming\\Lab2\\Lab2.sml";
[opening D:\Functional Programming\Lab2\Lab2.sml]
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
val reverse = fn : int list -> int list
val reverse' = fn : int list -> int list
val interleave = fn : int list * int list -> int list
val listToTree = fn : int list -> int tree
val revT = fn : int tree -> int tree
val trav = fn : int tree -> int list
val binarySearch = fn : int tree * int -> bool
val t = Node (Node (Node #,2,Node #),4,Node (Node #,6,Node #)) : int tree
val reverse_test = true : bool
val reverse'_test = true : bool
val interleave_test = true : bool
val listToTree_test = true : bool
val revT_test = true : bool
val binarySearch_test = true : bool
val it = () : unit
-
```

图 2.8 Lab2 测试结果

根据测试情况可知，Lab2 代码功能正确。

3 Lab 3

3.1 代码实现

3.1.1 编写函数 thenAddOne，类型为: ((int ->int) * int) -> int；功能为将一个整数通过函数变换(如翻倍、求平方或求阶乘)后再加 1

根据函数类型可知，thenAddOne 有两个参数：一个函数和一个整数。根据函数功能可知，首先对整数进行函数计算，然后计算结果加一。具体实现如下图：

```
(* thenAddOne: ((int -> int) * int) -> int
   REQUIRES: x is an integer
   ENSURES: f(x) + 1*)
fun thenAddOne (f, x) = f(x:int) + 1;
```

图 3.1 thenAddOne 函数的实现

3.1.2 编写函数 mapList，函数类型为: ((‘a -> ‘b) * ‘a list) -> ‘b list；功能为实现整数集的数学变换(如翻倍、求平方或求阶乘)

根据函数类型可知 mapList 有两个参数：一个函数和一个串。根据函数功能可知，需要依次从串中取出元素做函数操作，并组成新串。函数实现如下图所示：

```
(* mapList: ((‘a -> ‘b) * ‘a list) -> ‘b list
   REQUIRES: input is a list
   ENSURES: for each element in list: f(x)*)
fun mapList (f, []) = []
  | mapList (f, (x::L)) = f(x)::mapList(f, L);
```

图 3.2 mapList 函数的实现

3.1.3 编写函数 mapList’，函数类型为: (‘a -> ‘b) -> (‘a list -> ‘b list)，功能为实现整数集的数学变换(如翻倍、求平方或求阶乘)；比较函数 mapList’ 和 mapList，分析、体会它们有什么不同

根据题目可知，mapList’是一个多态函数，其本身计算的结果就是一个函数。对于 mapList’，同样需要依次从串中取出元素做函数操作，并组成新串。函数实现如下图所示：

```
(* mapList': ('a -> 'b) -> ('a list -> 'b list)
   REQUIRES: input is a list
   ENSURES: for each element in list: f(x)*)
fun mapList' f [] = []
  | mapList' f (x::L) = f(x)::mapList' f L;
```

图 3.3 mapList’ 函数的实现

3.1.4 编写函数 findOdd，要求：函数类型为: int list -> int option；功能为：如果 x 为 L 中的第一个奇数，则返回 SOME x；否则返回 NONE

根据题目要求，依次取出串中首元素进行判断是否为基数，若是返回 SOME x，否则递归检查下一个元素，直至串被取空，返回 NONE。具体实现如下图所示。

```
(* findOdd: int list -> int option
   REQUIRES: input is a list
   ENSURES: find the 1st odd num in list, return SOME X, else NONE*)
fun findOdd [] = NONE
  | findOdd (x::L) =
    let
        fun IsOdd 0 = false
          | IsOdd 1 = true
          | IsOdd x = IsOdd(x - 2)
    in
        case IsOdd x of false => findOdd L
                      | true => SOME x
    end
```

图 3.4 findOdd 函数的实现

3.1.5 编写函数 subsetSumOption: int list * int -> int list option，要求：对函数 subsetSumOption(L, s)：如果 L 中存在子集 L’，满足其中所有元素之和为 s，则结果为 SOME L’；否则结果为 NONE

对于传入的串 L，首先使用 subset 函数，得到类型为 int list list 的子串集合。对该集合，每次取出一个子集 L’，计算子集元素之和，与参数 2 进行大小比较，若相同则返回 SOME L’，若不同则递归比较下一个子集，直至没有子集剩余，返回 NONE。具体实现如下图：

```
(* subsetSumOption: int list * int -> int list option
   REQUIRES: input is a int list and an integer
   ENSURES: if exist a subset L' in L which the sum of all
            elements are equal to s, return SOME L', else NONE*)
fun subsetSumOption (L: int list, S: int) =
    let
        fun subset [] = [[]]
          | subset (x::L) =
            let
                val s = subset L
            in
                s @ mapList' (fn A => x::A) s
            end
        (* int int list * int -> int list option *)
        fun subsetsumcmp ([], _) = NONE
          | subsetsumcmp (L'::L, S) =
            let
            fun listsum [] = 0
            | listsum(x::L) = x + listsum L
            in
            case Int.compare(listsum L', S)
            of EQUAL => SOME L'
            | _ => subsetsumcmp(L,S)
            end
    in
        subsetsumcmp(subset L, S)
        (* first create a list to store all sublist of L
        then compare the sum of each sublist with S *)
    end
```

图 3.5 subsetSumOption 函数的实现

3.1.6 编写函数：exists: (‘a -> bool) -> ‘a list -> bool；forall: (‘a -> bool) -> ‘a list -> bool，对函数 p: t -> bool，整数集 L: t list,有：exist p L =>* true if there is an x in L such that p x=true；exits p L =>* false otherwise； forall p L =>* true if p x = true for every item x in L；forall p L =>* false otherwise

对于 exists 函数，每次从 list 中取出首元素比较是否满足要求，若满足则返回 true，否则递归检查下一个首元素，直至 list 为空，返回 false。

对于 forall 函数，每次从 list 中取出首元素比较是否满足要求，若不满足则直接返回 false，否则递归检查下一个首元素，直到 list 为空，返回 true。两个函数的具体实现如下图：

```
(* exists: ('a -> bool) -> 'a list -> bool
   REQUIRES: input is a list
   ENSURES: true if there is an x in L such that f x=true *)
fun exists f [] = false
  | exists f (x::L) =
    case f x
    of true => true
    | false => exists f L;

(* forall: ('a -> bool) -> 'a list -> bool
   REQUIRES: input is a list and it's not an empty one
   ENSURES: true if f x = true for every item x in L *)
fun forall f [] = true
  | forall f (x::L) =
    case f x
    of true => forall f L
    | false => false
```

图 3.6 exists 函数和 forall 函数的实现

3.1.7 编写函数：treeFilter: ('a -> bool) -> 'a tree -> 'a option tree，将树中满足条件 P（ 'a -> bool ）的节点封装成 option 类型保留，否则替换成 NONE

对传入的节点，若为空，则仍然得到空，若非空，则比较根节点 x 是否满足函数 f，若满足则将该节点的根节点修改为 SOME x，否则修改为 NONE。对左右子树重复该操作。具体代码实现如下图：

```
(* treeFilter: ('a -> bool) -> 'a tree -> 'a option tree
   REQUIRES: input is a tree
   ENSURES: for every node x satisfy f x, save it as SOME x, else NONE*)
fun treeFilter f Empty = Empty
  | treeFilter f (Node(L, x, R)) =
    case f x
    of true => Node(treeFilter f L, SOME x, treeFilter f R)
    | false => Node(treeFilter f L, NONE, treeFilter f R);
```

图 3.7 treeFilter 函数的实现

3.2 代码测试

编写测试代码如下图所示：

```
(* --------For Tests--------*)
fun double x = 2*x;
fun square x = x*x;
fun odd x =
    case x mod 2
    of 1 => true
    | _ => false
fun divisibleby3 x =
    case x mod 3
    of 0 => true
    | _ => false
val t:int tree = Node(Node(Node(Empty,1,Empty),2,Node(Empty,3,Empty)),4, Node(Node(Empty,5,Empty),6,Node(Empty,7,Empty)));
val test_thenAddOne =
    (thenAddOne (double, 7) = 15) andalso
    (thenAddOne (square, 8) = 65);
val test_mapList =
    (mapList (double, [1,2,3,4]) = [2,4,6,8]) andalso
    (mapList (square, [1,3,5,7]) = [1,9,25,49]);
val test_mapList' =
    (mapList' double [1,2,3,4] = [2,4,6,8]) andalso
    (mapList' square [1,3,5,7] = [1,9,25,49]);
val test_findOdd =
    (findOdd [2,4,6,7,1,5,2] = SOME 7) andalso
    (findOdd [2,4,6,8,10] = NONE) andalso
    (findOdd [] = NONE);
val test_subsetSumOption =
    (subsetSumOption([1,2,3,4,5], 16) = NONE) andalso
    (subsetSumOption([], 0) = SOME []) andalso
    (subsetSumOption([],5) = NONE) andalso
    (subsetSumOption([1,2,3,4,5,6], 16) = SOME [2,3,5,6]);
val test_exists =
    (exists odd [2,4,6,8,10] = false) andalso
    (exists divisibleby3 [2,4,6,8,10] = true) andalso
    (exists odd [1,2,3,4,5] = true);
val test_forall =
    (forall odd [1,2,3,4] = false) andalso
    (forall odd [1,3,85,71] = true) andalso
    (forall divisibleby3 [3,6,72,81] = true);
val test_treeFilter =
    (treeFilter odd t = Node(Node(Node(Empty,SOME 1,Empty),NONE,Node(Empty,SOME 3,Empty)),NONE, Node(Node(Empty,SOME 5,Empty),NONE,Node(Empty,SOME 7,Empty))))
    andalso
    (treeFilter divisibleby3 t = Node(Node(Node(Empty,NONE,Empty),NONE,Node(Empty,SOME 3,Empty)),NONE,Node(Node(Empty,NONE,Empty),SOME 6,Node(Empty,NONE,Empty))));
```

图 3.8 Lab3 测试代码

```
- use "D:\\Functional Programming\\Lab3\\Lab3.sml";
[opening D:\Functional Programming\Lab3\Lab3.sml]
datatype 'a option = NONE | SOME of 'a
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
val thenAddOne = fn : (int -> int) * int -> int
val mapList = fn : ('a -> 'b) * 'a list -> 'b list
val mapList' = fn : ('a -> 'b) -> 'a list -> 'b list
val findOdd = fn : int list -> int option
val subsetSumOption = fn : int list * int -> int list option
val exists = fn : ('a -> bool) -> 'a list -> bool
val forall = fn : ('a -> bool) -> 'a list -> bool
val treeFilter = fn : ('a -> bool) -> 'a tree -> 'a option tree
val double = fn : int -> int
val square = fn : int -> int
val odd = fn : int -> bool
val divisibleby3 = fn : int -> bool
val t = Node (Node (Node #,2,Node #),4,Node (Node #,6,Node #)) : int tree
val test_thenAddOne = true : bool
val test_mapList = true : bool
val test_mapList' = true : bool
val test_findOdd = true : bool
val test_subsetSumOption = true : bool
val test_exists = true : bool
val test_forall = true : bool
val test_treeFilter = true : bool
val it = () : unit
-
```

图 3.9 Lab3 测试结果

根据测试情况可知，代码功能正确。

4　实验心得

由于 ACM 班课程的原因，在大二上学期已经使用过 sml 语言，但是由于当时时间比较紧迫，对于 sml 语言的掌握只能说是一知半解，只求实现，不求得心应手。甚至由于当时的实验难度，一度对函数式编程感到畏惧。

但是在本学期选修函数式编程之后，终于有机会重新了解这门语言。通过回顾之前使用 sml 语言的经历，结合课上教授和课后实验的锻炼，终于克服了对于函数式编程的畏惧，并且能够熟练掌握函数式编程的技巧和思维方式，收获匪浅。

三次实验侧重各有不同，第一次实验主要侧重于熟练函数式编程的递归精神，第二次实验侧重于树结构，第三次实验侧重于多态函数和高阶函数，三次实验虽然难度不高，仅有几题稍要思考，但对提高学生面对陌生编程方式的信心大有裨益，同时也锻炼了学生函数式编程的能力。