



华中科技大学

函数式编程报告

姓 名：刘逸帆
学 院：计算机科学与技术学院
专 业：计算机科学与技术
班 级：校交 1601 班
学 号：U201610504

2019 年 5 月 6 日

目 录

1 课程内容梳理和总结	3
1.1 函数式编程定义	3
1.2 函数式编程解决了什么问题	3
1.3 函数式编程特点	3
1.4 函数式编程的正确性和有效性	5
1.5 函数式编程实际应用	6
2 实验问题分析和心得	7
2.1 实验目的	7
2.2 实验环境搭建	7
2.3 部分实验完成过程	11
3 课程学习体会和授课建议	18
3.1 课程学习体会	18
3.2 授课建议	18

1 课程内容梳理和总结

1.1 函数式编程定义

函数式编程是一种编程范式 (Programming paradigm)，它是一种构建计算机程序结构和元素的方式。函数式编程将计算机运算看作是数学中函数的计算，避免了状态以及变量的概念，即函数的返回值仅取决于其输入参数 (Indempotence)。函数式编程最重要的基础是 λ 演算 (lambda calculus)。

相比之下，命令式编程则具有副作用 (Side effect)，即同一表达式在不同时刻可能产生不同的结果。

1.2 函数式编程解决了什么问题

函数式编程与命令式编程最大的区别在于函数式编程避免了副作用，得益于这一本质区别，函数式语言避免了繁重的内存管理，为函数提供了引用透明性。

1.3 函数式编程特点

1. first-class values

程序、函数和过程可以使用数学意义下的函数来表示，换言之函数可以作为参数被传递到其他函数，如 $y = f(x)$ 。

2. 引用透明性 (Referential transparency)

函数式编程没有赋值语句，也就是说程序中的变量值一旦定义就不会改变。这消除了任何潜在的副作用。同样的，在函数式编程中任何函数的值只取决于它的参数的值，而与求得参数值的先后或调用函数的路径无关，这就是函数式编程的引用透明性。

得益于函数式编程的引用透明性，函数式程序中一般见到的多是表达式求值，这避免繁琐的内存管理，没有内部状态，也没有副作用。由于函数式编程的引用透明性，在函数式程序中：变量只是一个名称，而不是一个存储单元；由于变量只能被定义一次，在程序的执行过程中将反复拷贝大量数据。

3. 高阶函数 (Higher-Order Function)

高阶函数与 first-class values 密切相关，因为高阶函数和 first-class values 都允许函数作为参数和其他函数的结果。两者之间微妙的区别是：“高阶”描述的是对其他函数起作用的函数的数学概念，而“first-class”描述的是对其使用没有限制的编程语言实体的计算机科学术语。

函数式编程中，函数可以不依赖于任何其他对象而独立存在，可以将函数作为参数传入另一个函数，也可以作为函数的返回值。函数式编程中的函数由于具备这样的特性而被称为高阶函数。

高阶函数具有两个基本运算：合成和柯里化。

得益于高阶函数这一特性，函数的实用性被提高了，从而扩大了函数的适用范围，提前把易变因素传参固定下来；但也由于这样的特性，函数会延迟执行（惰性求值），这在某些应用场合中是必须的，但也会影响程序的执行效率。

4. 惰性求值（延迟计算）与并行

函数式编程的惰性求值特性指的是：当调用函数时，不是盲目的计算所有实参的值后再进入函数体，而是先进入函数体，只有当需要实参值时才计算所需的实参值（按需调用）。

基于这样的特性，在函数式编程中编译器能够用数学方法分析处理代码，从而能够：抵消相同项、避免执行无谓的代码；重新调整代码执行顺序、效率更高；重整代码以减少错误。但也使得程序不能处理 I/O，不能和外界交互

柯里化这一基本运算就具有延迟计算的特性，样例代码如下：

```
//未柯里化
func add1(x: Int, y: Int) -> Int {
    return x + y
}

//柯里化
func add2(x: Int) -> (Int) -> Int {
    return { $0 + x }
}
```

add1 和 add2 的例子展示了如何将一个接受多参数的函数变换为一系列只接受单个参数的函数，这个过程被称为柯里化；我们将 add2 称为 add1 的柯里化版本。

柯里化的优点：

(1) 提高适用性：保持函数参数的一致性有利于类型函数的组合。例如在高斯模糊的例子中，使用柯里化更有利于后期定义多滤镜的组合操作

(2) 延迟计算：就像 add(1)(2)一样，1 比 2 先传入，2 就会被延迟计算，在特定的场景里，有一定的应用意义。

(3) 参数复用：在高斯模糊的例子中，如果要使用相同的模糊半径对多张图片进行模糊操作时，可以复用模糊半径的参数

5. 递归调用及其优化

函数式语言中的迭代（循环）通常通过递归来完成。递归函数通常调用它们自己，从而让一个操作重复，直到它到达基本情况（Base case）。然而一般的递归操作需要栈结构，这十分容易空间浪费，并且可能由于堆栈溢出而引发异常。对此需要引入尾递归这一优化方式。

尾递归：一个函数(调用者,caller)调用另一个函数(被调用者, callee)，而且

callee 产生的返回值被 caller 立刻返回出去,这种形式的调用称为尾调用(tail call)。如果 caller 和 callee 是同一个函数,那么便将这个尾调用称为尾递归(tail recursion)。

6. 模式识别

模式: 只包含变量、构造子(数值、字符等)和通配符的表达式。

函数式编程语言倾向于使用类型化的 lambda 演算,在编译时拒绝所有无效程序并冒着误报错误,而不是无类型的 lambda 演算,这又被称为模式识别。

在程序执行的过程中,模式与值将进行匹配,如果匹配成功,将产生一个绑定(Binding);如果匹配不成功,就会失败并抛出异常。

例如:

模式 $d::L$ 和 $[2,4]$ 匹配的结果为: $d=2, L=[4]$

模式 $d::L$ 和 $[]$ 无法匹配,抛出异常

得益于模式识别这一特性,对于复杂的嵌套 if 语句,可以用更少的代码更好的完成任务;在修改代码时,如增加或修改条件,只需增加或修改合适的定义即可。

1.4 函数式编程的正确性和有效性

1. 函数式编程的正确性

以 Standard ML 语言为例,推荐在函数定义前,用注释信息描述函数功能,形如(* comments*):

函数名字和类型 (类型定义)

REQUIRES: 参数说明 (明确参数范围)

ENSURES: 函数在有效参数范围内的执行结果 (函数功能)

这样的代码说明有助于确保函数行为的正确性,同时确保在允许的参数范围内能得到正确的结果。

一般使用归纳法从理论上证明程序的正确性,归纳法又可进一步分为:简单归纳法 (simple (mathematical) induction)、完全归纳法 (complete (strong) induction)、结构归纳法 (structural induction)、良基归纳法 (well-founded induction) 等。对于不同类型的函数,分别对应有不同的归纳法适应该函数正确性的推导。

2. 函数式编程的有效性

程序有效性的分析主要涉及到程序近似执行时间的分析,在近似时间分析中,递推分析 (Recurrences) 是重要的分析手段之一。

递归函数的定义给出了程序的递推关系,执行情况用 work 表示, $W(n)$ 则表示参数规模为 n 的程序的执行情况 $work(W(n) = \text{work on inputs of size } n)$

$W(n)$ 的推导:

Base cases: 评估基本操作的执行 (Estimates the number of basic operations)

Inductive case: 用归纳法得到 $W(n)$ 的表达式 (Try to find a closed form solution)

for $W(n)$ using induction); 对表达式进行简化, 得到一个具有相同渐近属性的表达式(Find solution to a simplified recurrence with the same asymptotic properties)。

1.5 函数式编程实际应用

1. 工业领域

函数式编程长期以来在学术界流行, 但曾经几乎没有工业应用。然而, 最近几种主要的函数式编程语言已经在商业或工业系统中使用。例如, Erlang 编程语言由瑞典公司 Ericsson 在 20 世纪 80 年代后期开发, 最初用于实现容错电信系统。此后, 它已成为在北电、Facebook、法国电力公司和 WhatsApp 等公司编写一系列应用的热门语言。Lisp 的 Scheme 方言被用作早期 Apple Macintosh 计算机上的几个应用程序的基础语言, 最近已应用于诸如训练模拟软件和望远镜控制等场景。OCaml 于 20 世纪 90 年代中期推出, 已经在金融分析、驱动程序验证、工业机器人编程和嵌入式软件静态分析等领域得到了商业应用。Haskell 虽然最初是一种研究语言, 但也已被一系列公司应用于航空航天系统、硬件设计和网络编程等领域。

其他在工业中使用的函数式编程语言还有 Scala、F# (两者都是功能性 OO 混合, 支持纯函数和命令式编程)、Wolfram、Lisp、Standard ML 和 Clojure 等。

2. 教育领域

函数式编程被用作教授解决代数问题和几何概念的方法。它也在经典力学的结构和解释中被用作教授经典力学的工具。

2 实验问题分析和心得

2.1 实验目的

2.2.1 Lab-1

1. 熟悉 SML/NJ 开发环境及使用；
2. 掌握 SML 基本语法和书写规则；
3. SML 简单程序设计和程序编写。

2.2.2 Lab-2

1. 掌握 list 结构的 ML 编程方法和程序性能分析方法；
2. 掌握基于树结构的 ML 编程方法和程序性能分析方法。

2.2.3 Lab-3

1. 掌握多态类型、option 类型和高阶函数的编程方法；
2. 利用 ML 语言求解实际问题。

2.2 实验环境搭建

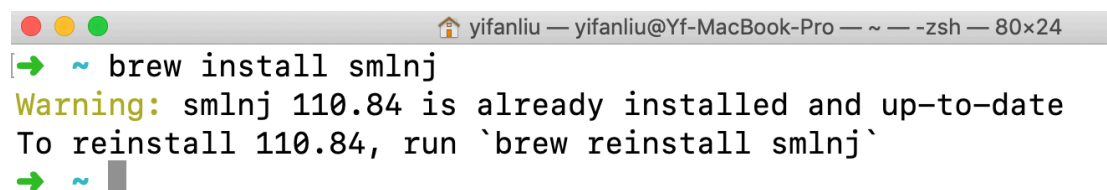
1. 系统环境

操作系统：MAC OSX

软件平台：sublime text 2

2. 搭建 SML 环境

(1) 在 OSX 系统的终端下通过 Home Brew 包管理器安装 smlnj，命令行如下：brew install smlnj，如图 2.1 所示。



```
yifanliu — yifanliu@Yf-MacBook-Pro — ~ — zsh — 80x24
➔ ~ brew install smlnj
Warning: smlnj 110.84 is already installed and up-to-date
To reinstall 110.84, run `brew reinstall smlnj`
➔ ~
```

图 2.1 使用 Home Brew 包管理器安装 smlnj

(2) 在软件 sublime text 2 中安装包管理器 Package Control，为了避免浏览境外网页时可能产生的不稳定，直接按照官网的手动安装教程为 sublime text 3 安装包管理器，网址如下：<https://packagecontrol.io/installation#st2>，手动安装教程详情如图 2.2 所示。

Manual

If for some reason the console installation instructions do not work for you (such as having a proxy on your network), perform the following steps to manually install Package Control:

1. Click the `Preferences > Browse Packages...` menu
2. Browse up a folder and then into the `Installed Packages/` folder
3. Download `Package Control.sublime-package` and copy it into the `Installed Packages/` directory
4. Restart Sublime Text

图 2.2 手动安装 sublime text 包管理器

(3) 在 sublime text 2 的菜单中选择 `Tools -> Command Palette` (快捷键 `Command+Shift+P`)，键入 `Package Control: Install Package`，随后选择分别键入 `"SML"` 和 `"SublimeREPL"` 以安装两个在 sublime text 中实现 SML 支持的包。若此时由于访问境外网站时效果不佳，可以直接前往对应 Package 的 Github 主页，参照对应的 Readme.md 文档进行操作 (“SML (Standard ML)”包的 Github 主页：<https://github.com/seanjames777/SML-Language-Definition>，对于每一个安装失败的包，可以通过观察 sublime text 中控制台输出的错误信息来找到该包的下载地址或项目名称)：通过 `git clone` 下载包文件后拖拽至 sublime text 的包路径即可。通过这种方式为 sublime text 添加包也同样适用于软件 sublime text 3，包安装结果如图 2.3 所示。

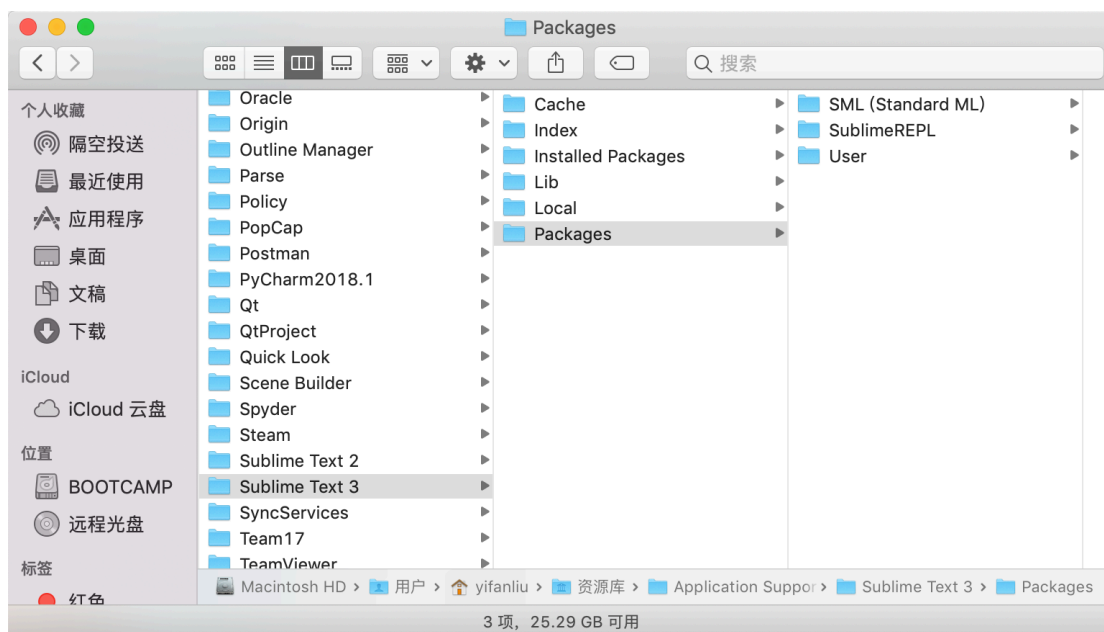


图 2.3 手动安装 SML 包

在 sublime text 中的包安装结果的视图如图 2.4 所示。

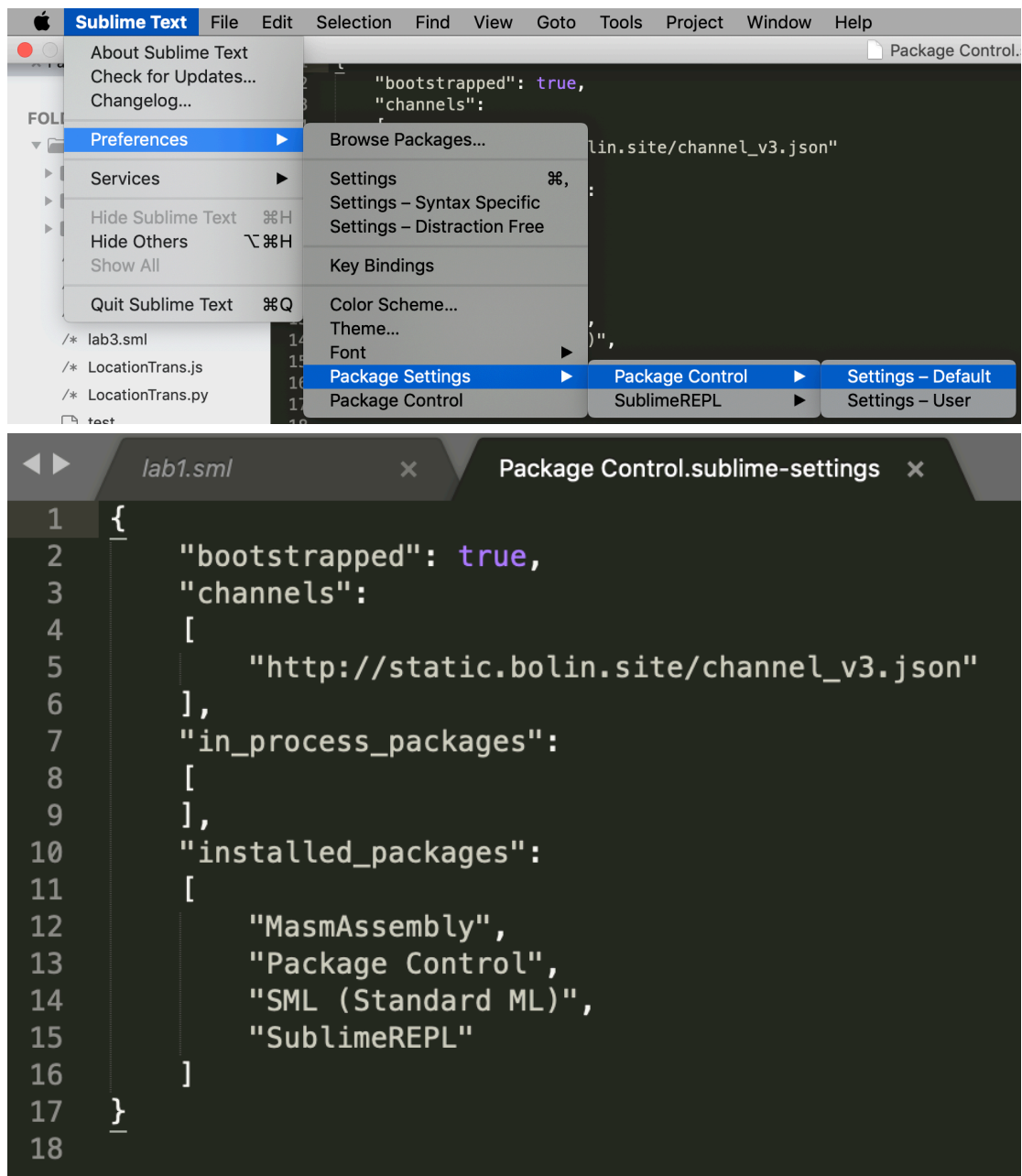


图 2.4 通过 sublime text 查看手动安装结果

重启 sublime text, 创建一个以.sml 后缀结尾的文件并在其中编写程序, 编写结束后通过 Command+B 快捷键即可编译程序并在控制台输出结果, 程序执行样例如图 2.5 所示。

```
test.sml x
1 (* SplitAt : int * tree -> tree * tree
2   REQUIRES: t is sorted
3   ENSURES: SplitAt(x,t) evaluates to a pair (t1,t2) of sorted trees
4             such that:
5             (a) for every Node(_,y,_) in t1, y <= x,
6             and (b) for every Node(_,y,_) in t2, y >= x,
7             and (c) trav(t1)@trav(t2) is a sorted permutation of trav(t).
8 *)
9 datatype tree = Empty
10 | Node of tree * int * tree;
11
12 fun SplitAt(x : int, Empty : tree) : tree * tree = (Empty, Empty)
13 | SplitAt(x, Node(left, y, right)) =
14   case Int.compare(x, y) of
15     LESS => let
16       val (t1, t2) = SplitAt(x, left)
17     in
18       (t1, Node(t2, y, right))
19     end
20   | _ => let
21     val (t1, t2) = SplitAt(x, right)
22   in
23     (Node(left, y, t1), t2)
24   end;
25
26 (*fun SplitAt(y, Empty) = (Empty, Empty)
27   | SplitAt(y, Node(t1, x, t2)) =
28     case Int.compare(x, y) of
29       GREATER => let
30         val (l1, r1) = SplitAt(y, t1)
31       in
32         (l1, Node(r1, x, t2))
33       end
34     => let
35       val (l2, r2) = SplitAt(y, t2)
36     in
37       (Node(t1, x, l2), r2)
38     end*)
39
40 val T = Node(Node(Empty,1,Empty),2,Node(Empty,3,Empty));
41 SplitAt(3,T);
```

```
find: /usr/local/smlnj*/bin: No such file or directory
Standard ML of New Jersey v110.84 [built: Fri Sep 28 19:40:37 2018]
[opening /Users/yifanliu/Documents/Code/sublimetext/test.sml]
datatype tree = Empty | Node of tree * int * tree
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[library $SMLNJ-BASIS/(basis.cm):basis-common.cm is stable]
[autoloading done]
val SplitAt = fn : int * tree -> tree * tree
val T = Node (Node (Empty,1,Empty),2,Node (Empty,3,Empty)) : tree
val it = (Node (Node (#,#,#),2,Node (#,#,#)),Empty) : tree * tree
-
```

图 2.5 使用 sublime text 编写并运行 SML 程序

另外在 sublime text 中运行的 sml 程序需要在每一句声明语句后以分号结尾，否则会编译失败。

2.3 部分实验完成过程

2.2.1 Lab-1

1. Question 5

题目及解题代码：

```
"----- Question 5 -----";
(**** 5 ****)
(*编写递归函数 square 实现整数平方的计算，即 square n = n * n。要求：程
序中可调用函数 double，但不能使用整数乘法（*）运算。*)
(* double : int -> int *)
(* REQUIRES: n >= 0 *)
(* ENSURES: double n evaluates to 2 * n. *)
fun double (0 : int) : int = 0
  | double n = 2 + double (n - 1);
fun square (0 : int) : int = 0
  | square n = if n > 0 then square(n-1) + double(n-1) + 1 else
               square(~n);
```

代码说明：设置递归终止条件 `square (0 : int) : int = 0`，递归逻辑为通过完全平方公式 $x^2 = (x-1+1)^2 = (x-1)^2 + 2*(x-1)*1 + 1$ 将待求的值展开，直至遇到递归终止条件；同时，在编写程序的过程中加入对负数的处理，因为是求取平方，直接将负数变换为正数再进行递归计算即可。

测试用例：

```
square 100;
square ~10;
```

测试结果：如图 2.6 所示。

```
val it = "----- Question 5 -----" : string
val double = fn : int -> int
val square = fn : int -> int
val it = 10000 : int
val it = 100 : int
```

图 2.6 测试用例输出结果

2. Question 6

题目及解题代码：

```
"----- Question 6 -----";
(**** 6 ****)
(*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'。
*)
```

```

(* 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 = true
  | divisibleByThree 1 = false
  | divisibleByThree 2 = false
  | divisibleByThree n = if n > 0 then divisibleByThree (n-3) else
                        divisibleByThree(~n);

```

代码说明：通过编写递归程序的方式实现模 3 运算，递归终止条件为输入数据被变换为 0、1、2 中的其中一个值，若是 0 说明输入数据能够被 3 整除，输出 true，否则输出 false；递归逻辑为对于输入数据递归减 3 直至到达终止条件，从而模拟除 3 运算的过程；同时，在编写程序的过程中加入对负数的处理，因为只需要判断输入值是否是 3 的倍数，直接将负数变换为正数再进行递归计算即可。

测试用例：

```

divisibleByThree 3;
divisibleByThree 2;
divisibleByThree ~6;
divisibleByThree ~8;
divisibleByThree 99;

```

测试结果：如图 2.7 所示。

```

val it = "----- Question 6 -----" : string
val divisibleByThree = fn : int -> bool
val it = true : bool
val it = false : bool
val it = true : bool
val it = false : bool
val it = true : bool

```

图 2.7 测试用例输出结果

2.2.2 Lab-2

1. Question 3

题目及解题代码：

```

"----- Question 3 -----";
(** 3 **)
(*编写函数 listToTree: int list -> tree，将一个表转换成一棵平衡树。*)
(*提示：可调用 split 函数，split 函数定义如下：
如果 L 非空，则存在 L1, x, L2，满足：

```

```

split L = (L1, x, L2) 且
L = L1 @ x :: L2      且
length(L1)和 length(L2)差值小于 1。*)
datatype tree = Empty
                | Node of tree * int * tree;
fun split ([] : int list) : (int list * int * int list) = ([], 0, [])
  | split (L : int list) : (int list * int * int list) =
    let val ll : int list = List.take (L, length L div 2)
        val mid : int = List.nth (L, length L div 2)
        val rl : int list = List.drop (L, length L div 2 + 1)
    in (ll, mid, rl)
    end;
fun listToTree ([] : int list) : tree = Empty
  | listToTree (L : int list) : tree =
    let val (ln, mid, rn) = split(L)
    in Node(listToTree(ln), mid, listToTree(rn))
    end;

```

代码说明：在本题中通过类型定义 `datatype` 来实现对二叉树结构的模拟，其中每个节点可能为一颗空树或是一棵树的根节点。在递归过程中通过 `let in` 语法调用 `split` 函数从而切分 `int list` 并将其转化为二叉树。

测试用例：

```
listToTree [1,3,5,7,9,10,11];
```

测试结果：如图 2.8 所示。

```

val it = "----- Question 3 -----" : string
datatype tree = Empty | Node of tree * int * tree
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[library $SMLNJ-BASIS/(basis.cm):basis-common.cm is stable]
[autoloading done]
val split = fn : int list -> int list * int * int list
val listToTree = fn : int list -> tree
val it = Node (Node (Node #,3,Node #),7,Node (Node #,10,Node #)) : tree

```

图 2.8 测试用例输出结果

2. Question 4

题目及解题代码：

```
"----- Question 4 -----";
```

```
(*** 4 ***)
```

(*编写函数 `revT: tree -> tree`，对树进行反转，使 `trav(revT t) = reverse(trav t)`。
(`trav` 为树的中序遍历函数)。*)

(*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）。*)

```
fun revT (Empty : tree) : tree = Empty
  | revT (Node(lt:tree, mid:int, rt:tree)) : tree = Node(revT rt, mid, revT lt);
```

代码说明：通过简单的递归逻辑，从二叉树的根节点开始，依次反转所有的左右子树。在将平衡二叉树反转后，通过编写中序遍历函数 `trav`，实现对转换结果的检验，即 `trav (revT T)`和 `reverse (trav T)`两条语句的输出应当相同。

测试用例：

```
val T : tree = Node (Node (Empty, 1, Empty),3 ,Node (Empty, 5, Empty));
revT T;
```

结果检查：

```
fun trav (Empty : tree) : int list = []
  | trav (Node(lt:tree, mid:int, rt:tree)) : int list = trav lt @ [mid] @ trav rt;
trav (revT T);
reverse (trav T);
```

测试结果：如图 2.9 所示。

```
val it = "----- Question 4 -----" : string
val revT = fn : tree -> tree
val T = Node (Node (Empty,1,Empty),3,Node (Empty,5,Empty)) : tree
val it = Node (Node (Empty,5,Empty),3,Node (Empty,1,Empty)) : tree
val trav = fn : tree -> int list
val it = [5,3,1] : int list
val it = [5,3,1] : int list
```

图 2.9 测试用例输出结果

3. Question 5

题目及解题代码

```
"----- Question 5 -----";
(** 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<, =, >。*)
(*datatype order = GREATER | EQUAL | LESS
case Int.compare(x1, x2) of
  GREATER => * x1 > x2 *
  | EQUAL => * x1 = x2 *
  | LESS => * x1 < x2 *)
fun binarySearch (Empty : tree, x : int) : bool = false
```

```
| binarySearch ((Node(lt:tree, value:int, rt:tree)) : tree, x : int) =
case Int.compare(value, x) of
    GREATER => binarySearch(lt, x)
    | EQUAL => true
    | LESS => binarySearch(rt, x);
```

代码说明：在搜索的过程中，通过 case 语句调用 Int.compare()，从而实现对两个输入值的大小的比较，并依据比较结果的不同产生不同的返回值。

测试用例：

```
binarySearch (T, 3);
binarySearch (T, 2);
```

测试结果：如图 2.10 所示。

```
val it = "----- Question 5 -----" : string
[autoloading]
[autoloading done]
val binarySearch = fn : tree * int -> bool
val it = true : bool
val it = false : bool
```

图 2.10 测试用例输出结果

2.2.3 Lab-3

1. Question 1

题目及解题代码：

```
"----- Question 1 -----";
(** 1 **)
(*编写函数 thenAddOne, 要求: *)
(*函数类型为: ((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。*)
fun double (0 : int) : int = 0
    | double n = 2 + double (n - 1);
fun square (0 : int) : int = 0
    | square n = if n > 0 then square(n-1) + double(n-1) + 1 else square(~n);
fun thenAddOne(f:int->int, x:int) = f(x) + 1;
```

代码说明：本次实验中涉及到了函数式编程的 first-class values 和高阶函数特性，即将函数作为参数传入函数，从而扩展了函数的功能。

测试用例：

```
thenAddOne(double, 10);
thenAddOne(square, 10);
```

测试结果：如图 2.11 所示。

```
val it = "----- Question 1 -----" : string
val double = fn : int -> int
val square = fn : int -> int
val thenAddOne = fn : (int -> int) * int -> int
val it = 21 : int
val it = 101 : int
```

图 2.11 测试用例输出结果

2. Question 3

题目及解题代码：

```
"----- Question 3 -----";
(** 3 **)
(*编写函数 mapList'， 要求： *)
(*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。*)
fun mapList' (f : 'a -> 'b) ([] : 'a list) : 'b list = []
  | mapList' (f : 'a -> 'b) (L : 'a list) : 'b list = mapList(f, L);
```

代码说明：本次实验中涉及到了函数式编程的延迟计算特性，即通过科里化运算触发了惰性求值的特性。以(mapList'(double)) ([1,3,5])为例，在函数实际执行的过程中，double 和[1,3,5]两个参数将会依次传入函数进行分析，这种特性也进一步扩展了函数的功能。

测试用例：

```
(mapList'(double)) ([1,3,5]);
(mapList'(square)) ([1,3,5]);
```

测试结果：如图 2.12 所示。

```
val it = "----- Question 3 -----" : string
val mapList' = fn : ('a -> 'b) -> 'a list -> 'b list
val it = [2,6,10] : int list
val it = [1,9,25] : int list
```

图 2.12 测试用例输出结果

3. Question 5

题目及解题代码：

```
"----- Question 5 -----";
(** 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。*)
fun subsetSumOption (L : int list, 0 : int) : int list option = SOME([])
  | subsetSumOption ([_] : int list, s : int) : int list option = NONE
```



```
| subsetSumOption (x::L : int list, s : int) : int list option =  
case subsetSumOption (L, s-x) of  
  NONE => subsetSumOption(L, s)  
  | SOME(l) => SOME(x::l);
```

代码说明：通过递归的做法来遍历寻找指定子串是一项比较困难的工作，在本题中采用在每次递归的过程中，排出子串的其中一边并修改需要满足的子串和这一做法来逼近寻找需要的子串。

测试用例：

```
subsetSumOption ([1,3,5,7],4);  
subsetSumOption ([1,3,5,7],12);
```

测试结果：如图 2.13 所示。

```
val it = "----- Question 5 -----" : string  
val subsetSumOption = fn : int list * int -> int list option  
val it = SOME [1,3] : int list option  
val it = SOME [5,7] : int list option
```

图 2.13 测试用例输出结果

3 课程学习体会和授课建议

3.1 课程学习体会

在刚开始进行实验时需要搭建对应的环境,在这项工作中由于不少需要使用到的境外网站访问不便,只能使用更偏向于手工的方式来搭建自己需要的语言环境。在这一部分的工作中我了解到了常用软件中各个 `Package` 的组织形式,具备了 DIY 一个更个性化的代码编辑工具的能力,收获颇丰。

以本次课程实验中 SML 语言为例的函数式编程为例,我深刻体会到了面向函数的编程方法,实验题目的难度由浅入深,十分锻炼思维。在整个实验流程过后我对函数式编程的皮毛有了大致了解,通过这门课程我也掌握了新的工具和编程思路,这对我以后的学习工作一定是会起到帮助的。

3.2 授课建议

对于同学来说,课堂作业和课后作业对当前知识的巩固作用都是很显著的,除了巩固之外也具有督促学习的作用,可以适当增加平时的练习量来优化授课效果。