华中科技大学

函数式编程原理课程报告

姓　　名：　　losyi

班　　级：　　CS2300

学　　号：　　U20231234

指导教师：　　顾琳

分数	
教师签名	

2025 年 11 月 12 日

目　录

一、 Heapify 求解

1.1 问题需求

实验要求实现一棵 min-heap 树，并实现 Heapify，用于将一棵任意二叉树转换为最小堆。

根据定义，一棵最小堆树 t 满足以下条件之一：

(1) t 是 Empty（空树）。

(2) t 是一个 Node(L, x, R)，其中 L 和 R 都是最小堆，并且 L 和 R 的根节点值（如果存在）都大于或等于 x。即 values(L) >= x 且 values(R) >= x。（其中 value(T) 函数用于获取树 T 的根节点的值）。

实验给出了三个函数定义需要我们实现：

(1) treecompare：当给定两棵树时，根据哪棵树的根节点值更大，返回 order 类型的值。

(2) SwapDown：确保 swapDown(t) = 如果 t 为空或 t 的所有直接子树都为空，则直接返回 t，否则返回一个包含 t 中所有元素的最小堆。

(3) heapify：给定任意树 t，计算结果为包含 t 中所有元素的最小堆。

1.2 思路与代码

二叉树的数据结构的定义实验已经给出，如下：

```
datatype tree = Empty | Br of tree * int * tree;
```

为了实现 heapify，需要先解决两个关键问题：

1. 如何比较子树的根节点值（用于判断是否需要调整节点位置）—— 对应 treecompare 函数；
2. 如何在子树已为堆的前提下，调整当前节点与子树的关系以满足堆性质 —— 对应 SwapDown 函数。

假设我们已经实现这两个函数，先对 heapify 进行设计，对于树 Br (l, x, r)，要使他成为堆，必须首先确保其子树 l 和 r 已经是堆，然后进行下沉（SwapDown）操作，而要使子树为堆，只需要递归调用自身即可。

基本情况：如果树 t 是 Empty，它本身已经是最小堆，直接返回即可。

递归步骤：对于树 Br (l, x, r)，首先要并行地递归调用 heapify 将左子树 l 和

右子树 r 转换成最小堆 heapL 和 heapR。此时得到的新树 Br (heapL, x, heapR)，其左右子树都满足最小堆性质，但根节点 x 可能不满足，即 x 可能大于 heapL 或 heapR 的根节点值。这就需要一个辅助函数来调整这种情况，也就是 SwapDown（下滤操作），通过将 x 下滤到正确位置，使整棵树成为最小堆。因此，heapify 的最后一步是调用 SwapDown (Br (heapL, x, heapR))。

由此可得代码如下：

```
(* heapify：将一棵任意二叉树转为最小堆 *)
fun heapify Empty = Empty
  | heapify (Br (l, x, r)) =
      let
          val heapL = heapify l
          val heapR = heapify r
      in
          SwapDown (Br (heapL, x, heapR))
      end;
```

treecompare 函数用于比较两棵树的根节点，在 SwapDown 中使用，设计较为简单，根据问题描述，我们需要处理四种情况。

(1) 当两棵树都是 Empty 时，它们相等，返回 EQUAL；

(2) 当左树为 Empty，右树非空时，定义空树小于任何非空树，返回 LESS；

(3) 当左树非空，右树为 Empty 时，返回 GREATER；

(4) 当两棵树都非空，即 Br (l1, x, r1) 和 Br (l2, y, r2) 时，直接使用 SML 内置的 Int.compare 比较它们的根节点值 x 和 y。

```
fun treecompare (Br (l1, x, r1), Br (l2, y, r2)) = Int.compare
 (x, y)
  | treecompare (Empty, Br(t1,x,t2)) = LESS
  | treecompare (Br(t1,x,t2), Empty) = GREATER
  | treecompare (Empty, Empty) = EQUAL;
```

SwapDown 是 heapify 的核心辅助函数，其前置条件严格限定输入树的左右子树已是最小堆，仅根节点可能“错位”（即根节点值大于子节点值）。函数的核心逻辑是“下滤”：将根节点与子树中较小的根节点比较，若根节点更大则交换位置，交换后新的子树可能破坏堆性质，需递归执行 SwapDown，直到根节点找到满足“小于等于所有直接子节点”的位置。其代码实现如下：

```
(* SwapDown：堆的下滤操作 *)
fun SwapDown Empty = Empty
```

```
  | SwapDown (Br (Empty, v, Empty)) = Br (Empty, v, Empty)
  | SwapDown (Br (Empty, v, Br (l2, y, r2))) =
      if v > y then Br (Empty, y, SwapDown (Br (l2, v, r2)))
      else Br (Empty, v, Br (l2, y, r2))
  | SwapDown (Br (Br (l1, x, r1), v, Empty)) =
      if v > x then Br (SwapDown (Br (l1, v, r1)), x, Empty)
      else Br (Br (l1, x, r1), v, Empty)
  | SwapDown (Br (Br (l1, x, r1), v, Br (l2, y, r2))) =
      if y < x andalso v > y then
        Br (Br (l1, x, r1), y, SwapDown (Br (l2, v, r2)))
      else if x <= y andalso v > x then
        Br (SwapDown (Br (l1, v, r1)), x, Br (l2, y, r2))
      else
        Br (Br (l1, x, r1), v, Br (l2, y, r2));
```

下面我们详细讲述各种情况，根据输入树的子树存在情况，分四种场景处理：

场景 1：空树（Empty）或叶节点（Br (Empty, v, Empty)）

空树本身是最小堆，直接返回 Empty；

叶节点无子女，天然满足“根节点大于等于子节点”（无子女即无约束），直接返回原叶节点 Br(Empty, v, Empty)。

场景 2：只有右子树（左空右非空：Br (Empty, v, Br (l2, y, r2))）

（1）比较当前根节点 v 与右子树根节点 y（因左子树为空，无需考虑左子树）；

（2）若 v ≤ y：右子树已是最小堆，当前根节点小于右子树根，整棵树满足堆性质，返回原树；

（3）若 v > y：交换 v 和 y，此时新的右子树为 Br(l2, v, r2)（原右子树的根变为 v）。由于 v 可能大于新右子树的子节点，需递归调用 SwapDown 处理新右子树，最终返回调整后的树 Br(Empty, y, SwapDown(Br(l2, v, r2)))。

场景 3：只有左子树（左非空右空：Br (Br (l1, x, r1), v, Empty)）

逻辑与“只有右子树”对称：

（1）比较当前根节点 v 与左子树根节点 x；

（2）若 v ≤ x，返回原树；

（3）若 v > x，交换 v 和 x，递归调用 SwapDown 处理新左子树 Br(l1, v, r1)，返回 Br(SwapDown(Br(l1, v, r1)), x, Empty)。

场景 4：左右子树均非空（Br (Br (l1, x, r1), v, Br (l2, y, r2))）

这是最复杂的场景，需先确定最小子节点，再与当前根节点比较。

第一步：找出左子树根 x 和右子树根 y 中的较小值（因左右子树已是堆，子树内最小值即为根节点）；

第二步：比较当前根 v 与较小子节点：

若 v≤ 较小子节点（即 v≤x 且 v≤y）：当前根节点满足堆性质，返回原树；

若 x 是较小子节点且 v > x：交换 v 和 x，新左子树为 Br(l1, v, r1)，递归处理新左子树，返回 Br(SwapDown(Br(l1, v, r1)), x, Br(l2, y, r2))；

若 y 是较小子节点且 v > y：交换 v 和 y，新右子树为 Br(l2, v, r2)，递归处理新右子树，返回 Br(Br(l1, x, r1), y, SwapDown(Br(l2, v, r2)))，图 1-1 展示下沉操作的过程。

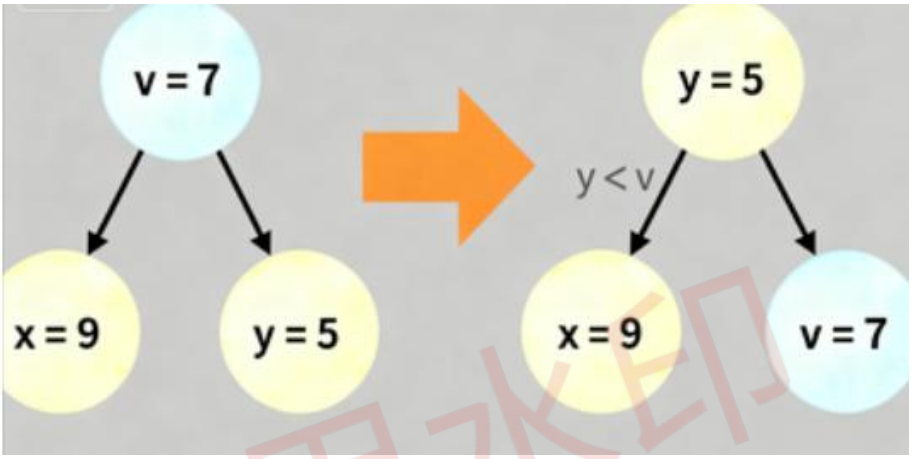


图 1-1 下沉操作示例

1.3 遇到的问题及运行结果

在实验过程中，遇到的问题主要来自 SwapDown 函数，起初，由于考虑不周到，出现了模式匹配不完整的问题，没有考虑左子树为空或者右子树为空的场景，缺少这两种模式会导致 SML 编译器发出“match nonexhaustive”警告。如果在运行时程序进入了这些未匹配的状态（例如，一个节点的左子树是 Empty 但右子树不是），程序将因未处理的异常而崩溃，如图 1-2 所示。

为解决 bug，我们需明确地为 SwapDown 添加两个新的模式匹配分支：

(1) SwapDown (Br (Empty, v, Br (l2, y, r2))) （处理只有右子树的情况）

(2) SwapDown (Br (Br (l1, x, r1), v, Empty)) （处理只有左子树的情况）

并在这两个分支中实现正确的比较和递归下滤逻辑。

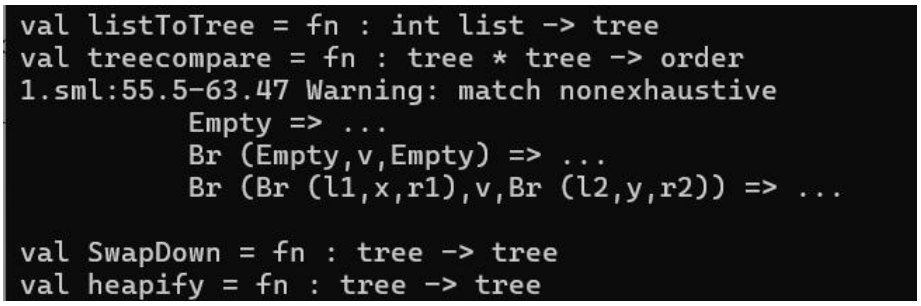


图 1-2 模式匹配不完整报错

另一个经典的错误是忘记递归下滤，例如，在处理左右子树都非空的场景时，当发现根节点 v 大于左子节点 x 时，只将 v 和 x 的位置交换，却没有对交换后根节点变为 v 的左子树继续调用 SwapDown。这种操作会破坏堆的性质，因为 v 被换下后，若仍大于其新的子节点，堆性质在更深层次依然不成立，v 必须下滤到最底部，或直到小于等于所有子节点为止。要解决这个问题，就需要在 SwapDown 中，每当 v 与子节点（x 或 y）交换位置后，对 v 进入的子树（即新的 Br (l1, v, r1) 或 Br (l2, v, r2)）递归调用 SwapDown，确保 v 能下沉到正确位置，图 1-3 展示出现该错误时的运行结果，可见中序遍历的结果不符合最小堆的性质。

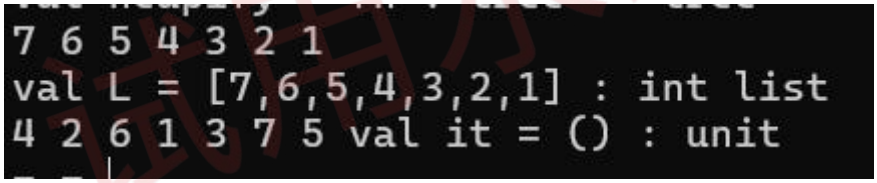


图 1-3 忘记递归调用的运行结果

如解题思路部分所示的代码对 bug 进行修复，在 SwapDown 中，每当 v 与子节点（x 或 y）交换位置后，必须对 v 进入的那个子树（即新的 Br(l1, v, r1) 或 Br(l2, v, r2)）递归地调用 SwapDown，以确保 v 能继续下沉到正确的位置。

测试输入 7 6 5 4 3 2 1，图 1-4 展示了正确的运行结果。

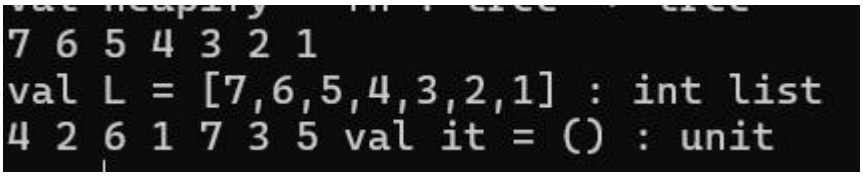


图 1-4 测试用例正确输出

1.4 性能分析

我们假设树是近似平衡的，即具有 n 个节点的树，其深度满足：

$$d \in O(\log n)$$

Work (W): 总工作量，即所有处理器执行的总操作数，相当于串行执行时间。

Span (S): 跨度，即最长依赖路径的长度，相当于在拥有无限处理器的情况下执行所需的时间。

1.4.1 SwapDown 复杂度分析

SwapDown 的执行路径是线性的。在每一步，它进行常数次比较和操作，然后最多对一个子树进行一次递归调用。

Work (W): 每次递归调用处理向下一层，工作量为 O(1)。这个过程最多持续 d 次（树的深度）。

$$W = O(\log n)$$

Span (S): 所有操作都是顺序的，没有可并行化的部分

$$S = O(\log n)$$

1.4.2 heapify 复杂度分析

heapify 在每个节点上执行操作。它首先处理子树，然后处理当前节点。

Work (W): heapify 访问树中的每个节点一次。对每个节点，heapify 都会：

递归处理左右子树，调用一次 SwapDown($O(\log n)$)

递推关系：

$$W(n) = W(n_L) + W(n_R) + O(\log n)$$

对于完全二叉树，这个递推求解得到：

$$W = O(n)$$

Span (S): 左右子树的 heapify 可以并行执行，假设树大致平衡，则 d_L 和 d_R 约等于$\log n - 1$。

$$S(\log n) \approx S(\log n - 1) + O(\log n)$$

展开这个递推式：

$$S(\log n) = O(\log n) + O(\log n - 1) + O(\log n - 2) + \ldots + O(1)$$

这是前 d 个整数的和，所以：Span = O(d^2)

对于平衡树 d 约为 O(log n)，因此最终结果如下表所示：

表 1-1 性能分析表

操作	Work（总工作量）	Span（跨度）
SwapDown	$(O(\log n))$	$(O(\log n))$
heapify	$(O(n))$	$(O((\log n)^2))$

1.5 高阶函数和多态类型

高阶函数，用于**同种数据的批量操作**，核心定义是能接收函数作为参数，或返回函数作为结果的函数，在 SML 中，**函数为一等公民**，这使得函数可被传递、赋值、嵌套组合，从根本上改变了代码的抽象方式。

在课件中，map 函数是一个典型的高阶函数

```
fun map f [] = []
  | map f (x::R) = (f x) :: (map f R)
```

其作用是：接受一个函数 f，并将它应用到列表中的每个元素上,类型签名为 ('a -> 'b) -> 'a list -> 'b list，意味着它不依赖具体元素类型，只需传入适配的转换函数（如 fn x => x*2 处理整数、fn s => String.size s 处理字符串），即可实现对任意类型列表的批量处理。

假设我们有一个将所有整数加 1 的函数，可以这样使用 map：

```
val add_one = fn x => x + 1
val result = map add_one [1, 2, 3]
```

在这个例子中，add_one 函数作为参数传给了 map，map 负责将 add_one 应用于列表 [1, 2, 3] 中的每个元素。

多态类型则允许一个函数适用于不同类型的数据。多态类型是类型变量的应用。例如，课件中有一个 split 函数，它可以拆分任何类型的列表：

```
fun split [] = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let val (A,B) = split L in (x::A, y::B) end;
```

split 函数的类型是 'a list -> 'a list * 'a list，这意味着 split 可以接收任何类型的列表（如 int list、string list 等）并返回两个同类型的子列表。

下面我们回顾下实验四中的题目，看看高阶和多态的妙用。

实验四第三关要求编写函数 mapList 和 mapList2,函数类型分别为: ((‘a -> ‘b) * ‘a list) -> ‘b list 和(‘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)。

```
fun mapList (f : 'a -> 'b, []) : 'b list = []
  | mapList (f, x::xs) = (f x) :: mapList (f, xs);
fun mapList2 (f : 'a -> 'b) : ('a list -> 'b list) =
    fn [] => []
     | (x::xs) => (f x) :: mapList2 f xs;
```

尽管存在语法差异，但 mapList 和 mapList2 均满足高阶函数与多态类型的核心定义，这是二者功能通用性的基础。在课件《函数式编程原理 Lecture 7》中指出：“高阶函数是以函数作为参数或返回函数作为结果的函数。”

这两个函数都接收函数 f : 'a -> 'b 作为参数，因此都属于高阶函数。

mapList：接收 (f, xs)，其中 f 是函数参数；

mapList2：不仅接收函数 f，还返回一个新的函数（即 'a list -> 'b list），能进一步被调用。

所以，mapList2 在“高阶性”上更典型——它既“接收函数”，又“返回函数”。这正对应课件中的 map 函数原型。

多态函数的核心判定标准是通过类型变量 'a、'b 适配多种类型，无需为每种类型重复定义，两类函数的类型签名均体现这一特征：

mapList：类型 ('a -> 'b) * 'a list -> 'b list 中，'a 表示“输入列表的元素类型”，'b 表示“转换后元素的类型”，可适配任意类型组合（如 int -> string、real -> bool 等转换函数）；

mapList2：类型 ('a -> 'b) -> ('a list -> 'b list) 同样包含 'a 和 'b 类型变量，与 mapList 具备完全相同的类型通用性。

两类函数的差异本质是函数抽象粒度的不同，导致适用场景的细微区别，这体现了函数式编程中参数传递灵活性的设计思想，mapList 和 mapList2 是“同一功能的不同语法实现”，核心差异源于柯里化设计带来的参数传递灵活性，而

共同的高阶函数与多态属性则确保了它们的通用性 —— 这也体现了 SML 作为函数式语言，在代码抽象与语法灵活上的设计优势。

通过对 mapList 和 mapList2 的分析，我们可以更直观地体会高阶和多态的特点与作用，下面进行一定总结。

高阶函数的关键在于函数也是数据。它们能够接收函数作为参数，也能返回函数作为结果。比如 mapList2 这一形式，就体现了这种思想：它先接受一个函数 f，再返回一个新的函数去处理列表中的每个元素。这种设计让计算逻辑被抽象出来，程序不再依赖具体的数据内容，而只描述做什么，而不是怎么做。通过这种抽象，代码可以更灵活地重用与组合。例如，我们只需定义一次 mapList2，就能让它批量应用任意函数——无论是对整数加一、对字符串求长度，还是对布尔值取反。

多态类型的作用则在于让这样的抽象能够跨越不同数据类型而保持类型安全。它通过类型变量（如 'a、'b）来表示泛化的类型，使一个函数能同时适用于多种数据。例如 mapList2 的类型是 ('a -> 'b) -> 'a list -> 'b list，意味着它能处理任何类型的列表，而不需要为每种类型单独编写版本。这种设计既提高了代码的通用性，也避免了类型错误在运行时才被发现的问题。

更重要的是，当高阶函数和多态类型结合在一起时，它们为程序提供了极强的抽象能力。高阶函数让逻辑更灵活可组合，多态类型让这些逻辑在不同数据结构上通用。两者共同的结果是，程序变得更加简洁、优雅和安全——只需少量定义，就能表达复杂的计算关系。这种编程方式体现了函数式语言的核心精神：用最小的语法结构实现最高层次的抽象。

二、 函数式拓展学习调研

2.1 函数式程序应用场景

函数式编程的核心思想在于“以函数为中心的计算”，强调**纯函数**、**不可变性**与**高阶抽象**。这些特性使它在许多需要高可靠性、高并发性或强逻辑表达能力的领域表现出独特优势。以下从三个典型场景出发——**并行计算**、**编译器与语言工具开发**、**金融与安全领域**——分析其应用与实际价值。

2.1.1 并行计算

在并行计算领域，函数式编程几乎是天然契合的范式。它的无副作用特性意味着每个函数的执行都不会依赖或修改外部状态，从而使得多个函数调用之间可以并发执行而不会发生数据竞争。这一特性使得函数式语言在处理大规模并行任务时更加安全可靠。

以 Erlang 为例，它被广泛应用于电信系统与实时服务器中。Erlang 的运行时系统支持上百万个轻量级并发进程，每个进程都是通过函数通信的形式进行交互，没有共享内存。这种模型完全基于函数式思想，使得并行计算的复杂度大大降低，系统的可扩展性与容错能力显著增强。

函数式编程还深刻影响了分布式计算框架的设计。Google 的 MapReduce 模型便直接借鉴了函数式中的 map 与 reduce 高阶函数：map 负责分布式地应用计算逻辑，reduce 负责汇总结果。该模型成为后续 Hadoop、Spark 等大数据框架的理论基础。如今，开发者通过定义纯函数即可在集群中并行处理海量数据，无需手动编写复杂的并发控制代码。这种简化不仅提高了开发效率，也显著减少了并发错误的风险。

综上，函数式编程以纯函数和不可变性为核心，使得并行计算的抽象更清晰、实现更安全，成为高性能计算和分布式系统中的重要理论支撑。

2.1.2 编译器与语言工具开发

函数式编程在编译器与语言工具开发领域有着深厚的传统与广泛应用。编译器本身就是一个需要复杂逻辑推理与数据转换的系统，而函数式语言在表达这种“输入—输出映射”关系时尤其自然。

以 **OCaml**、**Haskell** 和 **Scala** 为代表的函数式语言，在编译器实现中极具优势。OCaml 就是著名的 Coq 证明助手 与 Rust 编译器早期原型（rustc 0.x）的实现语言。OCaml 的模式匹配机制能够优雅地描述语法树的结构与转换规则，使得语法分析与语义推导过程更加直观。由于函数式编程强调不可变性与类型安全，编译器在优化与重构过程中可以更容易地保证正确性。

此外，函数式语言天然支持抽象语法变换与元编程。例如，Haskell 的高阶函数与 Monad 抽象为实现编译优化、静态分析提供了极强的表达能力。现代编译器工具（如 LLVM 的部分分析模块、Clang 插件系统）也借鉴了这种思想，通过纯函数式接口构建语义分析与代码生成流程。

在语言工具领域，函数式思想同样重要。例如，微软的 F# 编译器 完全采用函数式风格设计，其核心模块之间通过不可变数据流传递，显著提高了模块化程度和测试效率。由于函数式编程具备“引用透明性”，编译器的每个阶段都可以被单独测试或替换，而不会影响全局行为。

因此，函数式编程在编译器构建中不仅简化了逻辑实现，还提高了系统的稳定性与可维护性，为现代语言工具的可靠性和可扩展性奠定了坚实基础。

2.1.3 金融与安全领域

在金融和安全相关的系统中，函数式编程因其类型安全、可验证性强、错误率低等特性而被广泛采用。金融系统通常对正确性与稳定性要求极高，一旦出现计算错误或状态不一致，可能带来巨大的经济损失。函数式编程以其“纯函数 + 强类型系统”的特征，为这类高风险领域提供了稳固的技术保障。

在金融行业，**Haskell** 是使用最广泛的函数式语言之一。著名的投资银行和交易平台（如 Barclays、Standard Chartered）使用 Haskell 构建风险分析和衍生品定价系统。这些系统依赖高阶函数来描述复杂的金融模型和运算逻辑，而类型系统则在编译阶段确保所有数据流与计算公式严格匹配。例如，在利率衍生品定价中，通过函数式抽象可以清晰定义各类现金流、折现函数与收益计算，系统既灵活又安全。

函数式语言还适合构建形式化验证系统与区块链安全模型。在智能合约领域，Cardano 区块链平台采用 Haskell 与 Plutus（其函数式智能合约语言）来编写合约代码。纯函数与不可变状态使合约执行具有确定性和可验证性，能有效防止传统命令式语言中的副作用漏洞。这种函数式模型显著提高了合约的安全性和可审计性。

在网络安全方面，函数式语言的强类型系统和引用透明性为漏洞检测与安全验证提供了天然优势。程序员可以用代数数据类型（ADT）与模式匹配严格定义

协议与输入约束，编译器在类型推导阶段即可捕获潜在的逻辑错误。

总体来看，函数式编程在金融与安全领域的核心优势在于：它能将业务逻辑以数学形式精确表达，利用类型系统提前消除错误，并通过纯函数模型保证计算结果的可预测性。这使得函数式编程在高可靠性、高安全性的软件系统中占据重要地位。

2.2 函数式特征延伸

随着函数式编程理念的成熟与广泛传播，其核心特征已逐步被主流的非函数式语言（如 Java、C++ 等）吸收并融合。这些语言并未完全转向纯函数式范式，而是通过“选择性引入”关键概念，使程序在保持性能与兼容性的同时，获得函数式编程所带来的简洁性、安全性和可维护性。以下将从三个方面阐述这些函数式特征在非函数式语言中的体现与应用目的。

2.2.1 高阶函数与 Lambda 表达式

高阶函数是指能够将函数作为参数传递或作为返回值的函数。这一概念在函数式语言中十分常见，如 ML 语言中的 map 和 foldr 函数。Lambda 表达式则是匿名函数的简洁表示形式，可在需要时直接定义并传递函数逻辑，而无需单独命名函数。

自 Java 8 起，语言引入了 Lambda 表达式 与 Stream API，彻底改变了以往必须通过显式循环来操作集合的方式。程序员可利用函数式接口（如 Function<T, R>、Predicate<T>）配合 map、filter、reduce 等方法实现声明式的数据处理。例如：

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squares = nums.stream()
    .filter(x -> x > 2)
    .map(x -> x * x)
    .toList();
```

这段代码等价于传统的多层循环与条件判断，但语义更加清晰，且天然支持并行执行（通过 parallelStream()）。

在 C++11 中，Lambda 表达式同样成为核心语言特性。例如：

```
std::vector<int> nums = {1, 2, 3, 4};
std::vector<int> squares(nums.size());
```

```
std::transform(nums.begin(), nums.end(), squares.begin(),
               [](int x) { return x * x; });
```

这里的 std::transform 与函数式语言中的 map 作用类似，Lambda 表达式将函数逻辑直接嵌入算法调用中。

高阶函数与 Lambda 表达式的引入，使传统语言具备了更高层次的抽象能力，并强化了数据流式处理和函数组合的能力。

2.2.2 多态类型

多态类型是函数式编程的重要特征之一，指函数或数据结构可以在多种数据类型下通用，而无需为每种类型单独定义实现。其核心思想是在类型系统中引入类型变量（如 'a、'b），在编译时再进行具体化。

Java 通过 泛型 实现类型参数化。例如：

```
public static <T> void printList(List<T> list) {
    for (T element : list)
        System.out.println(element);
}
```

该函数可接受任意类型的列表（如 List<Integer> 或 List<String>），与函数式语言中的 'a list -> unit 类型模式相似。

C++ 则通过模板实现类似功能：

```
template <class T, class F>
std::vector<T> map(std::vector<T> xs, F f) {
    std::transform(xs.begin(), xs.end(), xs.begin(), f);
    return xs;
}
// C++20 Concepts 约束示例（可选）
template <class T> concept Addable = requires(T a, T b){ a + b;
 };
template <Addable T> T add(T a, T b){ return a + b; }
```

通过多态，提升了代码的通用性、复用性与类型安全性。通过将具体类型抽象为参数，开发者可以编写一次逻辑、适用于多种数据类型，从而显著减少重复代码。同时，泛型或模板在编译期完成类型检查和实例化，既避免了运行时类型转换的开销，又防止了类型错误，增强了程序的健壮性。此外，结合高阶函数（如上述 C++ 中的 map 模板），多态类型还支持构建高度灵活且类型安全的函数组合管道，为声明式编程提供了坚实基础。

2.2.3 Option/Optional

ML 语言使用 option 数据类型（datatype 'a option = NONE | SOME of 'a） 来显式处理一个值可能缺失的情况。例如课件中的 mkchange 函数，它要么返回 NONE（找不到解），要么返回 SOME A（找到解 A）。

Java 8 引入了 java.util.Optional<T> 类

```
int age = repo.findById(id)
              .map(User::getAge)
              .filter(a -> a >= 0)
              .orElse(0);
```

C++17 引入了 std::optional<T> 类

```
std::optional<User> u = repo.find(id);
int age = u ? u->age() : 0;  // 或 u.value_or(0)
```

通过引入这种特性（约束），增了强程序的健壮性，避免空指针错误。在 Java 和 C++ 的传统实践中，null 常被用来表示“值的缺失”，但这极易导致空指针异常这一常见的运行时错误。Optional 将“可能缺失”这一概念纳入了类型系统，强制程序员显式地检查值是否存在（例如使用 .isPresent()）或提供默认值，从而在编译层面就避免了 null 相关的运行时错误。这符合函数式编程利用静态类型检测提供运行时保障的特点。

2.2.4 Python 中的函数式特性

Python 作为一种多范式语言，从早期版本开始就深度融合了函数式编程的特性。它将函数视为“一等公民”，允许函数被自由传递和赋值。

高阶函数与 Lambda：Python 内置了 map()、filter() 以及 functools.reduce() 等高阶函数，它们的功能与 ML 中的 map 和 foldr 一致。这些函数通常与 lambda 匿名函数结合使用，以快速定义内联操作。

列表推导式：这是 Python 中最具代表性的函数式特性之一。它提供了一种极其简洁的语法，将 map 和 filter 的功能合二为一。

```
# Java Stream: nums.stream().filter(x -> x > 2).map(x -> x * x)
.toList()
# Python 等价实现：
nums = [1, 2, 3, 4, 5]
squares = [x * x for x in nums if x > 2]
```

生成器与惰性求值：Python 的生成器实现了惰性求值。这意味着数据项只在被迭代时才逐个产生，而不是一次性在内存中创建整个集合，这对于处理大规模或无限数据集至关重要。

Python 引入这些特性的目的同样是为了简化数据转换和处理，用更具声明性、可读性更高的代码替代传统的 for 循环。

综上所述，Java、C++ 和 Python 等主流开发语言纷纷引入函数式编程特征，其根本目的在于解决现代软件开发中的核心痛点。这些优势可以总结为以下几点：

(1)**提升代码简洁性与可读性**：函数式特征提供了声明式的编程风格。无论是 Java 的 Stream API、C++ 的 std::transform 还是 Python 的列表推导式，它们都用简洁的链式调用或单行表达式替代了繁琐的命令式循环和条件分支，使代码意图更加清晰。

(2)**增强程序的健壮性与安全性**：Optional / option 类型的引入，将“可能缺失的值”纳入类型系统管理 ，强制开发者显式处理 null 或 NONE 的情况，从而在编译期（或通过静态分析）规避了大量的空指针运行时错误 。泛型和模板提供的多态性，确保了算法在不同类型上操作时的类型安全，避免了不安全的类型转换。

(3)**提高代码的抽象与复用性**：高阶函数允许将“行为”参数化，使算法与具体的操作逻辑解耦 。多态类型则允许将“数据类型”参数化，使同一套算法逻辑可以复用于多种数据结构 。

(4)**易于并行处理与优化**：函数式代码天然具有无副作用和数据不变性的特点。这使得 map、filter 等操作可以被轻松地并行化，因为它们不依赖共享状态或修改外部数据，从而简化了并发编程的复杂度。

最终，这些函数式特征帮助开发者编写出更可靠、更易维护、更具表现力且更易于优化的现代软件。

2.3 高阶与多态函数

在 1.5 中，我已经举了很多例子说明高阶与多态函数的特征和我的理解，这里我再举几个课件中的例子进一步说明。

foldr 函数,见图 2-1 介绍，类型为('a * 'b -> 'b) -> 'b -> 'a list -> 'b，其中：

高阶特征：参数 F 是一个二元函数（类型为'a * 'b -> 'b），foldr 通过 F 实现对列表元素的聚合计算；

多态特征：类型变量'a 支持任意类型的列表元素，'b 支持任意聚合结果类型（如 int real string）。

多态函数：foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

- 函数功能：

  for all types t_1, t_2,

  all n≥0, and all values F: t_1 * t_2 -> t_2, [x_1,...,x_n] : t_1 list, z : t_2 ,

  foldr F z [x_1,...,x_n] = F(x_1, F(x_2, ..., F(x_n, z)...))

```
fun foldr F z [ ] = z
  | foldr F z (x::L) = F(x, foldr F z L)
```

图 2-1 多态函数示例 foldr

foldr 的核心目的是 “对列表元素进行批量聚合操作”（如求和、求最大值、字符串拼接），通过抽象聚合逻辑（函数 F）和初始值（z），实现对不同聚合需求的通用支持。例如：

int list 求和：聚合函数 F 使用 op +（即整数加法函数），初始值 z=0，代码如下：

```
fun sum L = foldr (op +) 0 L
sum [1,2,3,4] =>* 1 + (2 + (3 + (4 + 0))) = 10
```

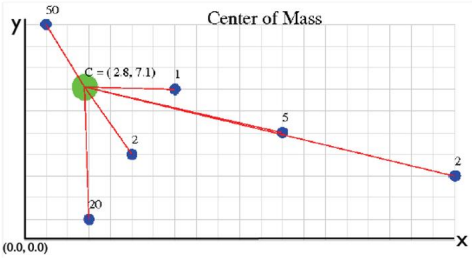
real list 求最大值：聚合函数 F 使用 Real.max（实数最大值函数），初始值 z 为列表第一个元素，代码如下：

```
fun maxlist (x::R) = foldr Real.max x R
maxlist [3.2, 1.5, 4.7, 2.9] =>* Real.max(3.2, Real.max(1.5, R
eal.max(4.7, 2.9))) = 4.7
```

此处 foldr 的多态性切换为支持 real list 类型，高阶性通过 Real.max 实现最大值聚合，体现灵活的批量处理能力。

下面我们看另一个实例，求解点集中心，见图 2-2.

高阶函数应用2——求解点集中心



给定点集：[(m_1,(x_1,y_1)),...,(m_n,(x_n,y_n))]

求解中心点(X, Y): real * real，满足

X = (m_1*x_1 + ... + m_n*x_n)/M

Y = (m_1*y_1 + ... + m_n*y_n)/M

M = m_1 + ... + m_n

图 2-2 高阶函数应用实例-求解点集中心

该实例的求解逻辑完全依赖 map（批量转换数据）和 foldr（批量聚合数据）两个高阶函数，二者均满足 “以函数为参数” 的核心特征，且在课件中均有明确定义：

map 函数：类型为('a -> 'b) -> 'a list -> 'b list，功能是将参数函数批量应用于列表中每个元素，返回转换后的新列表。

foldr 函数：类型为('a * 'b -> 'b) -> 'b -> 'a list -> 'b，功能是通过参数函数（聚合逻辑）和初始值，对列表元素从右至左递归聚合，返回最终聚合结果。

```
type point = real * real
type body = real * point
(* 辅助函数：点坐标加法（用于聚合坐标） *)
fun add((x1,y1), (x2,y2)):point = (x1+x2, y1+y2)
(* 辅助函数：提取 body 中的质量 m（用于转换数据） *)
fun mass (m, (x, y)) = m
(* 辅助函数：计算单个 body 的“质量×坐标”（用于转换数据） *)
fun scale r (m, (x, y)) = (r * m * x, r * m * y)
(* 核心求解函数：点集中心 *)
fun center (L : body list) : point = let
    (* 步骤 1：用 map 提取所有 body 的质量，再用 foldr 聚合总质量 M *)
    val M = foldr (op +) 0.0 (map mass L)
    (* 步骤 2:用 map 计算每个 body 的“(m×x)/(M)、(m×y)/(M)”,再用 foldr 聚合总坐标和 *)
    val total = foldr add (0.0, 0.0) (map (scale (1.0/M)) L)
in
    total  (* 步骤 3：返回聚合后的中心坐标(X,Y) *)
end
```

该实例虽未直接定义新的多态函数，但核心依赖 map 和 foldr 的多态特性，实现对不同类型列表的通用处理，具体体现为：

map 的多态性：在实例中两次复用 map，但处理的列表类型完全不同：

第一次：map mass L，输入列表类型为 body list，输出列表类型为 real list（依赖 map 的类型变量'a=body，'b=real）；

第二次：map (scale (1.0/M)) L，输入列表类型仍为 body list，输出列表类型为 point list（依赖 map 的类型变量'a=body，'b=point）。

若没有多态特性，需为 “body list 转 real list” “body list 转 point list” 分别编写两个遍历函数，导致代码冗余。

foldr 的多态性：同样两次复用 foldr，聚合的列表类型和结果类型不同：

第一次：foldr (op +) 0.0 (map mass L)，输入列表类型为 real list，聚合结果类型为 real（依赖 foldr 的类型变量'a=real，'b=real）；

第二次：foldr add (0.0,0.0) (map (scale (1.0/M)) L)，输入列表类型为 point list，聚合结果类型为 point（依赖 foldr 的类型变量'a=point，'b=point）。

多态特性使 foldr 既能处理 “实数求和”，又能处理 “坐标求和”，无需为不同聚合场景编写专用聚合函数。

由此可见，高阶函数通过 “函数作为参数 / 返回值” 实现批量逻辑复用，多态函数通过 “类型变量” 实现多类型通用处理，两者结合是函数式编程灵活性的核心来源。从课件中的 map foldr pair 等实例可见，其核心目的均为减少重复代码、提升代码可维护性与通用性，避免为不同数据类型或不同处理逻辑编写冗余的遍历 / 聚合代码，这也是函数式编程在复杂数据处理场景中高效性的关键。

三、 函数式编程学习建议与心得

坦白地说，最初选修函数式编程，是因为学长们评价“作业少，没考试”的建议，并不明白什么是函数式编程，而在学习 ML 语言时，更是被各种语法和模式匹配弄得头晕，在进一步的学习和实践过后，我才逐渐领会函数式编程的精髓，重点不在于学习其语法，而是它其中逻辑化，数学化的思维方式。

函数式编程的理论根源来自 λ 演算与图灵机模型的对立：前者以函数为核心，强调表达式求值与引用透明性，而后者依赖状态与命令操作。这种根本差异，使函数式编程成为用数学方法思考程序的实践载体。

初学函数式编程，最大的障碍在于思维模式的转变。对于熟悉 C、Java 等命令式语言的学习者而言，放弃循环、变量和赋值操作，改用递归与模式匹配来表达计算逻辑，是一种根本的思维重构。课程中讲到的 sum 与 fib 函数示例让我深有体会：

```
fun sum’ ([ ], a) = a | sum’ (x::L, a) = sum’ (L, x+a);
fun fib 0 = 0 | fib 1 = 1 | fib n = fib(n-2) + fib(n-1);
```

这些函数通过递归定义自然地表达了“分解问题—组合结果”的思想，而无需显式的循环控制或状态维护。进一步的**尾递归优化**也让我认识到函数式语言在性能层面同样可以高效。

通过课程学习，我体会到函数式编程的最大魅力在于抽象与安全。在实验三实现 heapify，我可以很容易地根据报错发现模式匹配不完整的问题，在设计相关函数时，我不必去仔细思考一步步下沉，转化的过程，只需从宏观上，逻辑上进行设计与分析，后面，我们又学习了高阶，多态等内容。高阶函数实现了对“行为”的抽象，多态类型实现了对“数据”的抽象，而模式匹配则实现了对“结构”的抽象。例如课件中的通用排序 Msort，通过将比较函数 cmp 作为参数传入，实现了对任意类型排序的泛化。

因此，函数式编程的学习重点不在于语法，或是那些传统的算法，而是对思维方式的重塑。它教会我们用数学逻辑去理解程序结构，用类型系统去约束行为，用抽象与组合去提升可重用性，用函数式编程去解决我们曾经学过的问题（如排序问题，最小堆问题），带给我们另一种解决问题的角度，这些收获远超语言本身。