

函数式编程原理

课程报告

**姓 名： 李一帆**

**学 号： U201811495**

**班 级： 校交1802**

**指导教师： 顾琳**

**计算机科学与技术学院**

**2021 年 4 月28 日**

一、函数式语言家族成员调研

完成函数式语言家族成员调研报告，内容可包括但不限于：庞大的函数式语言家族中有哪些成员？都由谁提出来的？各自有什么特征？没落和兴盛的原因？

**1、函数式语言家族成员：**

主要可以分为纯函数式编程语言、非纯函数式编程语言。

其中，纯函数式编程语言语言通常不允许直接使用程序[状态](https://zh.wikipedia.org/w/index.php?title=%E7%8A%B6%E6%80%81_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)&action=edit&redlink=1)以及[易变对象](https://zh.wikipedia.org/wiki/%E4%B8%8D%E5%8F%AF%E8%AE%8A%E7%89%A9%E4%BB%B6)，主要有：Miranda、Clean、Haskell;

非纯函数式编程语言可以分为静态类型和动态类型。

静态类型的主要有：ML、OCaml、F#、Scala等

动态类型主要有：Lisp、Scheme、Clojure、Racket、LOGO、Erlang、Wofram、R。

**2、函数式语言发展历史：**

（1） 20世纪50年代后期，[John McCarthy](https://zh.wikipedia.org/wiki/John_McCarthy_(computer_scientist)" \o "John McCarthy (computer scientist))开发了早期的函数式语言[**LISP**](https://zh.wikipedia.org/wiki/LISP)，运行在大型IBM主机上。LISP的函数是使用邱奇的lambda表示法定义的，并扩展了标签构造来允许[递归](https://zh.wikipedia.org/wiki/%E9%80%92%E5%BD%92_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)" \o "递归 (计算机科学))函数。最开始的LISP是多[范型](https://zh.wikipedia.org/wiki/%E7%BC%96%E7%A8%8B%E8%8C%83%E5%9E%8B" \o "编程范型)语言，并且随着新的范型的发展，越来越多的编程风格得到了支持。

（2）后来发展出来的方言比如[Scheme](https://zh.wikipedia.org/wiki/Scheme" \o "Scheme)、[Clojure](https://zh.wikipedia.org/wiki/Clojure" \o "Clojure)，和分支语言比如[Dylan](https://zh.wikipedia.org/wiki/Dylan_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Dylan (编程语言))和[Julia](https://zh.wikipedia.org/wiki/Julia_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Julia (编程语言))等，试图简化LISP，使它围绕一个函数式核心，而[Common Lisp](https://zh.wikipedia.org/wiki/Common_Lisp" \o "Common Lisp)旨在保留并更新它所替代的各种更早先LISP方言的那些范型特征。

（3）1956年发明的**[IPL](https://zh.wikipedia.org/wiki/%E4%BF%A1%E6%81%AF%E5%A4%84%E7%90%86%E8%AF%AD%E8%A8%80" \o "信息处理语言)**语言，一般被认为是第一个基于计算机的函数式编程语言。它是一种用于操纵符号列表的汇编式语言。它有一个生成器的概念，相当于一个接受函数作为参数的函数，并且，由于它是汇编级语言，代码可以是数据，因此IPL可以被视为具有[高阶函数](https://zh.wikipedia.org/wiki/%E9%AB%98%E9%98%B6%E5%87%BD%E6%95%B0" \o "高阶函数)。但是，它在很大程度上依赖于改变列表的结构和类似的指令式编程特征。

（4）20世纪60年代早期，[Kenneth E. Iverson](https://zh.wikipedia.org/wiki/Kenneth_E._Iverson" \o "Kenneth E. Iverson)开发了**[APL](https://zh.wikipedia.org/wiki/APL%E8%AF%AD%E8%A8%80" \o "APL语言)** ，在他1962年出版的《A Programming Language》一书中有介绍。 APL给John Backus的[FP](https://zh.wikipedia.org/wiki/FP_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "FP (编程语言))提供了巨大的影响。

（5）20世纪90年代早期，Iverson和Roger Hui创造了**[J](https://zh.wikipedia.org/wiki/J%E8%AF%AD%E8%A8%80" \o "J语言)**语言。

（6）20世纪90年代中期，以前曾与Iverson合作过的Arthur Whitney创建了**[K](https://zh.wikipedia.org/wiki/K_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "K (编程语言))**语言，后者在金融行业中与其派生出来的[Q](https://zh.wikipedia.org/w/index.php?title=Q_(%E9%98%B5%E5%88%97%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)&action=edit&redlink=1)语言一起被商业化使用。

（7）1977年[John Backus](https://zh.wikipedia.org/wiki/John_Backus" \o "John Backus)在他的图灵奖颁奖演讲《可以从冯·诺依曼式的编程风格中解放出来的程序设计和函数式风格及其程序代数》中，展示了他提出的**[FP](https://zh.wikipedia.org/wiki/FP_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "FP (编程语言))**。他将函数式编程定义为通过“组合形式”以分层方式构建，允许“程序代数”; 在现代语言中，这意味着函数式程序应遵循[复合性原理](https://zh.wikipedia.org/wiki/%E5%A4%8D%E5%90%88%E6%80%A7%E5%8E%9F%E7%90%86" \o "复合性原理)。Backus的论文推广了函数式编程的研究，虽然它强调的是[函数级编程](https://zh.wikipedia.org/wiki/%E5%87%BD%E6%95%B0%E7%BA%A7%E7%BC%96%E7%A8%8B" \o "函数级编程)而不是现在所说的lambda演算风格。

（8）1973年[爱丁堡大学](https://zh.wikipedia.org/wiki/%E7%88%B1%E4%B8%81%E5%A0%A1%E5%A4%A7%E5%AD%A6" \o "爱丁堡大学)的[Robin Milner](https://zh.wikipedia.org/wiki/Robin_Milner)发明了**[ML](https://zh.wikipedia.org/wiki/ML%E8%AF%AD%E8%A8%80" \o "ML语言)**语言，它的语法受到了[ISWIM](https://zh.wikipedia.org/wiki/ISWIM" \o "ISWIM)的启发。

（9）1973年，David Turner在[圣安德鲁斯大学](https://zh.wikipedia.org/wiki/%E5%9C%A3%E5%AE%89%E5%BE%B7%E9%B2%81%E6%96%AF%E5%A4%A7%E5%AD%A6" \o "圣安德鲁斯大学)开发[**SASL**](https://zh.wikipedia.org/wiki/SASL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80))语言，它基于了[ISWIM](https://zh.wikipedia.org/wiki/ISWIM" \o "ISWIM)的应用式子集。在1976年，Turner重新设计并重新实现它为[惰性求值](https://zh.wikipedia.org/wiki/%E6%83%B0%E6%80%A7%E6%B1%82%E5%80%BC" \o "惰性求值)语言。

（10）在20世纪70年代的爱丁堡，Burstall和Darlington开发了**[NPL](https://zh.wikipedia.org/wiki/NPL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "NPL (编程语言))**语言。NPL基于[Kleene递归方程](https://zh.wikipedia.org/w/index.php?title=Kleene%E9%80%92%E5%BD%92%E6%96%B9%E7%A8%8B&action=edit&redlink=1) ，并在他们的程序转换工作中首次引入。

（11）然后Burstall、MacQueen和Sannella结合了来自ML的多态类型检查，从[NPL](https://zh.wikipedia.org/wiki/NPL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "NPL (编程语言))派生出了**[Hope](https://zh.wikipedia.org/wiki/Hope_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Hope (编程语言))**语言。

（12）ML最终发展成几种语言，其中最常见的是**[OCaml](https://zh.wikipedia.org/wiki/OCaml" \o "OCaml)**和**[Standard ML](https://zh.wikipedia.org/wiki/Standard_ML" \o "Standard ML)**。

（13）在20世纪70年代，[Guy L. Steele](https://zh.wikipedia.org/wiki/Guy_L._Steele" \o "Guy L. Steele)和[Gerald Jay Sussman](https://zh.wikipedia.org/wiki/Gerald_Jay_Sussman" \o "Gerald Jay Sussman)开发了**[Scheme](https://zh.wikipedia.org/wiki/Scheme" \o "Scheme)**，如有影响力的“Lambda论文集”和经典的1985年教科书《[计算机程序的构造和解释](https://zh.wikipedia.org/wiki/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A8%8B%E5%BA%8F%E7%9A%84%E6%9E%84%E9%80%A0%E5%92%8C%E8%A7%A3%E9%87%8A" \o "计算机程序的构造和解释)》中所描述的那样。[Scheme](https://zh.wikipedia.org/wiki/Scheme)是使用[词法作用域](https://zh.wikipedia.org/wiki/%E8%AF%8D%E6%B3%95%E4%BD%9C%E7%94%A8%E5%9F%9F" \o "词法作用域)和[尾调用优化](https://zh.wikipedia.org/wiki/%E5%B0%BE%E8%B0%83%E7%94%A8)的第一个Lisp方言，将函数式编程的影响力提升到更广泛的范围，让更多的编程语言社区接触到它们。

（14）在20世纪80年代，[佩尔·马丁-洛夫](https://zh.wikipedia.org/wiki/%E4%BD%A9%E5%B0%94%C2%B7%E9%A9%AC%E4%B8%81-%E6%B4%9B%E5%A4%AB" \o "佩尔·马丁-洛夫)开发了[直觉类型论](https://zh.wikipedia.org/wiki/%E7%9B%B4%E8%A7%89%E7%B1%BB%E5%9E%8B%E8%AE%BA)（也称为构造类型论），它将函数式编程与表现为类型依赖的数学证明联系起来。这导致了[交互式定理证明](https://zh.wikipedia.org/w/index.php?title=%E4%BA%A4%E4%BA%92%E5%BC%8F%E5%AE%9A%E7%90%86%E8%AF%81%E6%98%8E&action=edit&redlink=1)的新方法的产生，并影响了后续的函数式编程语言的发展。

（15）David Turner开发的惰性求值函数式语言**[Miranda](https://zh.wikipedia.org/wiki/Miranda_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Miranda (编程语言))**最初出现在1985年，采用来自[ML](https://zh.wikipedia.org/wiki/ML%E8%AF%AD%E8%A8%80" \o "ML语言)与[Hope](https://zh.wikipedia.org/wiki/Hope_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "Hope (编程语言))语言的概念，作为他先前所设计的[SASL](https://zh.wikipedia.org/wiki/SASL_(%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80)" \o "SASL (编程语言))和[KRC](https://zh.wikipedia.org/wiki/%E8%82%AF%E7%89%B9%E9%80%92%E5%BD%92%E8%AE%A1%E7%AE%97%E5%99%A8" \o "肯特递归计算器)语言的后继者。

（16）Miranda对后来的**[Haskell](https://zh.wikipedia.org/wiki/Haskell" \o "Haskell)**有很强的影响，由于它是专有软件，所以Haskell社区于1987年开始达成共识，以形成函数式编程研究的开放标准，对标准的实现自1990年以来一直在进行中。

1. 上机实验心得体会

1、在熟悉安装SML/NJ开发环境的基础上，完成实验内容。在完成实验中选取2-3个感受最深的实验，谈谈在函数式编程学习和实践中的心得和体会。

**心得体会：**

本次函数式编程课主要是学习了一种与C/C++等我们之前学习过的语言不同的一种思维模式。其实函数式的思想理论上可以在大一的时候上计算思维、导论之类的课程涉及到了，很久以前翻阅《计算机程序的构造和解释》(SICP)这本传说中的计算机圣经（现在很少有人提到这本，主要都是推荐去看CSAPP了）的时候，作者使用的貌似就是一种古老的函数式的语言，虽然当时只看了一部分，但是通过函数式的思想描述一些算法让人感觉是一件很自然的事情，比较符合人们的思维方式。

这学期的函数式编程课我主要还是通过CS15150的课程PPT以及lecture notes，还有网站上给的几个SML学习链接和PSML这本书（看了一部分）来学习。为什么除了上课外没有细看老师的PPT？因为感觉英文原版的notes和SML的指南讲的更清楚，课上听老师讲了哪些部分的内容，课下去看看相关的具体语法和推导性的证明就能理解了。

2、针对本课程的内容设置、实验设置等，提出宝贵的建议和意见。

**关于实验设置的建议：**

可能因为课时较短，实验内容确实有点简单。

建议可以分为语法自测和函数思想和数据结构几个部分的实验。

语法自测，可以设置包含能实现大多数功能的函数的语法，让学生快速熟悉SML（或者其他语言）的语法；

数据结构：能够SML语言自定义数据结构，个人认为这部分的实验较少

函数思想：要求实现的函数都蛮简单，可以尝试实现稍复杂的函数，给出部分框架。

推导：统一推导格式，增加几个题目。

其他上课讲过的一些模块可以加入提高版本的实验内容。

具体可以参见cs15150的实验和作业。

**三、实验记录**

这部分是实验中的一些记录，因为实验已检查过代码和其思想，这里只是给出了部分实验内容。

实验一：

一、课堂作业检测

1、

|  |  |
| --- | --- |
| 3 + 4; | val it = 7 : int |
| 3 + 2.0; | operator and operand don’t agree |
| it+6; | val it = 13 : int |
| val it = “hello”; | val it = “hello” : string |
| it+”world”; | overloaded variable not defined at type |
| it+5; | overloaded variable not defined at type |
| val a = 5; | val a = 5 : int |
| a = 6; | val it = false : bool |
| a + 8; | val it = 13 : int |
| val twice = (fn x => 2 \* x); | val twice = fn : int -> int |
| twice a; | val it = 10 : int |
| let val x = 1 in x end; | val it = 1 : int |
| foo; | unbound variable or constructor: foo |
| [1, “foo”]; | operator and operand don’t agree |

2、

|  |  |
| --- | --- |
| x::L | 成功 |
| \_::\_ | 成功 |
| x::y(::L) | 当list 长度>=2时，成功 |
| (x::y)::L | 失败，类型 int list list |
| [x, y] | 当list 长度=2时，成功 |

3、

|  |  |
| --- | --- |
| list of length 3 | [x, y, z] |
| lists of length 2 or 3 | 无，需要拆为两个模式 |
| Non-empty lists of pairs | (x, y)::L |
| Pairs with both components being non-empty lists | (x::L1, y::L2) |

4、

|  |  |  |
| --- | --- | --- |
|  | 类型 | 值 |
| 第4行的x | int | 2 |
| 第5行的m | real | 12.4 |
| 第6行的x | int | 9001 |
| 第14行的assemble(x, 3.0)计算结果 | int | 27 |

5、

（1）zip

（2） unzip

不成立

6、

fun circ (r : real) : real = 2.0 \* pi \* r;

fun semicirc : real = pi \* r;

fun f(3:int) : int = 9

| f \_ = 4;

fun area (r : real) real = pi \* r \* r;

二、代码

(\* lab 1-2\*)

(\* 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);

(\* Testcase \*)

val 120 = mult [1, 2, 3, 4, 5];

val 1 = mult [];

val 2 = mult [1, 2];

(\* lab 1-3\*)

(\* 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);

(\* Testcase \*)

val 120 = Mult [[1, 2], [3, 4], [5]];

val 1 = Mult [];

val 1 = Mult [[], []];

val 1 = Mult [[1], [1]];

val 2 = Mult [[1], [2]];

(\* lab 1-4-1 \*)

(\* mult': int list \* int -> int

REQUIRES: true

ENSURES: mult'(L, a) evaluate to the product of the integers in L and a

\*)

fun mult' ([], a) = a

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

(\* Testcase \*)

val 120 = mult' ([1, 2, 3, 4, 5], 1);

val 1 = mult' ([], 1);

val 2 = mult' ([1, 2], 1);

(\* lab 1-4-2 \*)

(\* Mult': int list list \* int -> int

REQUIRES: true

ENSURES: Mult'(R, a) evaluate to the product of al the intergers in the lists of R and a

\*)

fun Mult'([], a) = a

| Mult' (r::R, a) = Mult'(R, 1) \* mult'(r, a);

(\* Testcase \*)

val 120 = Mult' ([[1, 2], [3, 4], [5]], 1);

val 1 = Mult' ([], 1);

val 1 = Mult' ([[], []], 1);

val 1 = Mult' ([[1], [1]], 1);

val 2 = Mult' ([[1], [2]], 1);

(\* lab 1-5 \*)

(\* 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);

(\* square: int -> int

REQUIRES: n >= 0

ENSURES: square n evaluates to n \* n

\*)

fun square (0 : int): int = 0

| square(x) = square(x - 1) + double(x - 1) + 1;

(\* Testcase \*)

val 1 = square 1;

val 36 = square 6;

(\* lab 1-6 \*)

(\* 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 x = divisibleByThree(x - 3);

(\* Testcase \*)

val false = divisibleByThree 1;

val false = divisibleByThree 2;

val true = divisibleByThree 3;

val true = divisibleByThree 6;

val false = divisibleByThree 7;

(\* lab 1-7 \*)

(\* oddP: int -> bool

REQUIRES: n >= 0

ENSURES: evenP n evaluates to true iff n is odd

\*)

fun oddP (0: int): bool = false

| oddP 1 = true

| oddP n = oddP (n - 2);

(\* Testcase \*)

val false = oddP 2;

val true = oddP 1;

val true = oddP 19;

val false = oddP 20;

实验二：

1、

|  |  |
| --- | --- |
| all | fn: int \* string list -> string list |
| your | int |
| base | string list |
| funny | fn : (‘a \* int -> int) \* ‘a list -> int |
| F | fn: ‘a \* int -> int |
| X | ‘a |
| xs | ‘a list |
| 表达式 | fn: ‘b -> string |

2、

Theorem: for all sorted lists L, ins(x, L) = a sorted permutation of x::L

Proof: By induction on length of L

Base case: When L has length 0, L must be [], [] is sorted. Show ins(x, []) = a sorted permutation of x::[]

Inductive step: prove for L = x::R

Inductive hypothesis: let k > 0 and assume IH:

For all sorted lists A of length < k, ins(x, A) = a sorted permutation of x::A.

Let L be a sorted list of length k

Showing:

pick y and R such that L = y::R., Length(R) < k

so that R is a sorted list with length < k, and y <= all of R

for the first case, ins(x, R) = a sorted permutation of x::R, because compare(x, y) = GREATER, y::ins(x, R) = a sorted permutation of x::L

for the second case, ins(x, y::R) = a sorted permutation of x::(y::R)

Theorem: for all values L, isort(L) = a sorted permutation of L

Proof: By induction on length of L

Base case: for L = [], isort [] = a sorted permutation of []

Inductive step: prove for L = x::R

Inductive hypothesis:

For all lists L of length < k, isort(L) = a sorted permutation of L

To show: isort(x::L) = a sorted permutation of x::L

Showing:

Let L be a list length of k

Pick y and R such that L = y::R, length(R) < k

So that R is a sorted list with length < k

By IH, Isort R = a sorted permutation of R

Because ins(y, R) = a sorted mermutation of y::R

Isort(y::R) = a sorted permutation of L

3、

O(2^n)

O(n)

二、实验代码

(\* lab 2-1-1\*)

(\* reverse int list -> int list

REQUIRES: true

ENSURES: reverse(L) reverse a list without helper function

\*)

fun reverse ([]) : int list = []

| reverse (x::L : int list) = reverse (L) @ [x];

(\* Testcase \*)

val [5, 4, 3, 2, 1] = reverse [1, 2, 3, 4, 5];

val [] = reverse [];

(\* lab 2-1-2\*)

(\* reverse\_helper int list \* int list -> int list

REQUIRES: true

ENSURES: store result of reversed L into RL

\*)

fun reverse\_helper ([], RL) = RL

| reverse\_helper (x::L, RL) = reverse\_helper (L, x::RL);

(\* reverse' int list -> int list

REQUIRES: true

ENSURES: reverse'(L) reverse a list with helper function, complexity O(n)

\*)

fun reverse' (L: int list) : int list = reverse\_helper (L, []);

(\* Testcase \*)

val [5, 4, 3, 2, 1] = reverse' [1, 2, 3, 4, 5];

val [] = reverse' [];

(\* lab 2-2 \*)

(\* interleave: int list \* int list -> int list

REQUIRES: true

ENSURES: merge two list one by one

\*)

fun interleave ([], LB : int list) : int list = LB

| interleave (LA : int list, []) = LA

| interleave (x::LA, y::LB) = x::y::interleave(LA, LB);

(\* Testcase \*)

val [2, 4] = interleave([2], [4])

val [2, 4, 3 ,5] = interleave([2, 3], [4, 5]);

val [2, 4, 3, 5, 6, 7, 8, 9] = interleave([2, 3], [4, 5, 6, 7, 8, 9])

val [2, 3] = interleave([2, 3], [])

datatype tree = Empty | Node of tree \* int \* tree;

(\* Ins: int \* tree -> tree

REQUIRES: true

ENSURES: insert a node x to a sortedTree

\*)

fun Ins(x, Empty) = Node(Empty, x, Empty)

| Ins(x, Node(left, root, right)) =

case Int.compare(x, root) of

LESS => Node(Ins(x, left), root, right)

| \_ => Node(left, root, Ins(x, right));

(\* trav: tree -> int list

REQUIRES: true

ENSURES: traverse a tree in inorder

\*)

fun trav (Empty) = []

| trav (Node(left, root, right)) = trav(left) @ (root::(trav(right)));

(\* lab 2-3 \*)

(\* split:

REQUIRES: true

ENSURES:split L to (L1, x, L2) with L = L1 @ x::L2 and |length(L1) - length(L2)| < 1

\*)

fun split ([]) = ([], 0, [])

| split ([x]) = ([], x, [])

| split (x::L) =

let

val (left, root, right) = split L

in

if length(left) > length(right)

then (left, x, root::right)

else (root::left, x, right)

end;

(\* listToTree: int list -> tree

REQUIRES: true

ENSURES: converse a tab to a AVL tree

\*)

fun listToTree ([]) = Empty

| listToTree ([x]) = Node(Empty, x, Empty)

| listToTree (L) =

let

val (left, root, right) = split (L)

in

Node(listToTree(left), root, listToTree(right))

end;

(\* Testcase \*)

val list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

val tree1 = listToTree list1;

trav tree1;

(\* lab 2-4 \*)

(\* revT: tree -> tree

REQUIRES: a binary ALV tree

ENSURES: trav(revT t) = reverse(trav t)

\*)

fun revT (Empty) = Empty

| revT (Node(left, root, right)) = Node(revT (right), root, revT (left));

(\* Testcase \*)

trav(revT tree1) = reverse(trav tree1);

(\* lab 2-5 \*)

(\* binarySearch: tree \* int -> bool

ENQUIRES:

ENSURES:

\*)

fun binarySearch (Empty, \_) = false

| binarySearch (Node(left, root, right), x) =

case Int.compare(x, root) of

LESS => binarySearch(left, x)

| EQUAL => true

| GREATER => binarySearch(right, x);

(\* Testcase \*)

fun createSortedTree [] = Empty

| createSortedTree (x::L) = Ins(x, createSortedTree(L));

val sortedTree = createSortedTree ([9, 8, 7, 6, 5, 4, 3, 2, 1]);

val false = binarySearch(sortedTree, 0);

val true = binarySearch(sortedTree, 9);

val true = binarySearch(sortedTree, 4);

val false = binarySearch(sortedTree, 10);

实验三：

第五题的work和span：

在一个深度为d的树上

d = 0时，Wswapdown(d) = k0，

d > 0时，Wswapdown(d) = k1 + Wswapdown(d-1)

WswapDown(d) = O(d)

d = 0时,Wheapify(d) = k0，

d > 0 时，Wheapify(d) = k1 + 2Wheapify(d-1) + Wswapdown(d)

Wheapify(d) = O(d2^d)

d = 0，Sswapdown(d) = k0

d > 0，Sswapdown(d) = k1 + Sswapdown(d-1)

Sswapdown(d) = O(d)

d = 0时，Sheapify(d) = k2

d > 0时，Sheapify(d) = k3 + Sheapify(d-1) + Sswapdown(d)

Sswapdown(d) = O(d^2)

二、实验代码

(\* lab 3-1 \*)

(\* thenAddOne: ((int -> int) \* int) -> int

REQUIRES: true

ENSURES: evaluate a integer to function f then add one

\*)

fun thenAddOne (f : int->int, x : int) : int = f (x) + 1;

(\* Testcase \*)

val square = fn (x : int) => x \* x;

val 17 = thenAddOne (square, 4);

val double = fn (x : int) => 2 \* x;

val 9 = thenAddOne (double, 4);

(\* lab 3-2-1 \*)

(\* mapList: (('a -> 'b) \* 'a list) -> 'b list

REQUIRES: true

ENSURES: map function f to a list

\*)

fun mapList (f, []) = []

| mapList (f, x :: L) = f (x) :: mapList (f, L);

(\* Testcase \*)

val [2, 4, 8] = mapList (double, [1, 2, 4]);

val [1, 4, 16] = mapList (square, [1, 2, 4]);

(\* lab 3-2-2 \*)

(\* mapList': ('a -> 'b) -> ('a list -> 'b list)

REQUIRES: true

ENSURES: map function f to a list

\*)

fun mapList' (f) =

fn (L) => case L of

[] => []

| x::R => f (x) :: (mapList' f) R;

(\* Testcase \*)

val [2, 4, 8] = (mapList' double) [1, 2, 4];

val [1, 4, 16] = (mapList' square) [1, 2, 4];

(\* lab 3-3 \*)

(\* findOdd: int list -> int option

REQUIRES: true

ENSURES: if x is the first odd in L, return SOME x; else return NONE;

\*)

fun findOdd ([]) = NONE

| findOdd (x::L) =

if (x mod 2) = 1 then SOME x

else findOdd L;

(\* Testcase \*)

val SOME 1 = findOdd ([1, 2, 3, 4, 5]);

val NONE = findOdd ([2, 4, 6, 8, 10]);

(\* lab 3-4 \*)

(\* treeFilter: ('a -> bool') -> 'a tree -> 'a option tree

REQUIRES: true

ENSURES: replace the node in tree with option

\*)

datatype 'a tree = Empty | Node of 'a tree \* 'a \* 'a tree;

fun treeFilter (f, Empty) = Empty

| treeFilter (f, Node(left, root, right)) =

if (f (root)) then

Node(treeFilter (f, left), SOME root, treeFilter (f, right))

else

Node(treeFilter (f, left), NONE, treeFilter (f, right));

(\* Testcase \*)

val aboveZero = fn x => x > 0;

val tree1 = Node(Node(Empty, 5, Empty), ~3, Node(Empty, 1, Empty));

val tree2 = Node(Node(Empty, SOME 5, Empty), NONE, Node(Empty, SOME 1, Empty))

val tree2 = treeFilter (aboveZero, tree1);

(\* lab 3-5 \*)

datatype tree = Node of tree \* int \* tree | Empty

(\* treecompare: tree \* tree -> order

REQUIRES: true

ENSURES: when given two trees, returns a value of type order,

based on which tree has a largervalue at the root node

\*)

fun treecompare(Empty : tree, Empty : tree) : order = EQUAL

| treecompare(Empty, \_) = GREATER

| treecompare(\_, Empty) = LESS

| treecompare(Node(\_, root1, \_), Node(\_, root2, \_)) = Int.compare(root1, root2);

(\* Testcase \*)

val GREATER = treecompare(Node(Empty, 5, Empty), Node(Empty, 2, Empty))

val LESS = treecompare(Node(Empty, 2, Empty), Node(Empty, 6, Empty))

val EQUAL = treecompare(Node(Empty, 3, Empty), Node(Node(Empty, 5, Empty), 3, Empty))

(\* SwapDown: tree -> tree

REQUIRES: the subtrees of t are both minheaps

ENSURES: swapDown(t) = if t is Empty or all of t’s immediate children are empty then

just return t, otherwise returns a minheap which contains exactly the elements in t.

\*)

fun swapDown (Empty : tree) : tree = Empty

| swapDown(T as Node(left, root, right)) =

let

val (smallTree, bigTree) =

case treecompare(left, right) of

GREATER => (right, left)

| \_ => (left, right)

in

case smallTree of

Empty => Node(smallTree, root, bigTree)

| Node(smallLeft, smallRoot, smallRight) =>

(case treecompare(T, smallTree) of

GREATER => Node(swapDown(Node(smallLeft, root, smallRight)), smallRoot, bigTree)

| \_ => T)

end;

(\* Testcase \*)

val Node(Node(Empty, 3, Empty), 2, Node(Empty, 4, Empty)) =

swapDown(Node(Node(Empty, 4, Empty), 3, Node(Empty, 2, Empty)))

val Empty = swapDown(Empty)

(\* heapify : tree -> tree

REQUIRES: tree

ENSURES: given an arbitrary tree t, evaluates to a minheap with exactly the elements of t.

\*)

fun heapify(Empty : tree) : tree = Empty

| heapify(Node(left, root, right)) = swapDown(Node(heapify(left), root, heapify(right)));

val Node(Node(Empty, 4, Empty), 2, Empty) = heapify(Node(Node(Empty, 2, Empty), 4, Empty))

val Empty = heapify Empty