Principles of Programming

(4190.210)

Chung-Kil Hur (허충길)

Department of Computer Science and Engineering

Seoul National University

Syllabus

➢Lecture

- Tue & Thu: 17:00 ~ 18:15 (@302-107)
- https://github.com/snu-sf-class/pp202402
- Bring your laptop to lectures

➢Instructor

- Chung-Kil Hur
- http://sf.snu.ac.kr/gil.hur/

➢Teaching Assistant

- Yeji Han & Yeonwoo Nam
- Contact TAs at pp@sf.snu.ac.kr

➢Grading (tentative)

- Attendance: 5%
- Assignments: 25%
- Midterm exam: 30%
- Final exam: 40%

Who am I?

- Prof. Chung-Kil (Gil) Hur [허충길]
  - Education: KAIST (B.S.), Univ of Cambridge (Ph.D.)
  - (High school) Bronze medal in IMO 1994 @Hongkong.
  - Software Foundations Lab
    http://sf.snu.ac.kr
  - Research Topics
    - Software Verification
    - Low-level Language Semantics (C/C++/LLVM/Rust)
    - Relaxed-Memory Concurrency
  - Our collaborators
    - Many in USA, UK, Germany, France, Israel.
  - Publications in Programming Languages (PL) area
    - POPL and PLDI are the two most prestigious conferences in PL area (see https://csrankings.org)
    - Published 12 PLDI and 8 POPL papers
    - Received distinguished paper awards from PLDI 2017, 2021 & POPL 2020
    - General Chair for PLDI 2025 in Seoul

Introduction

Overview

- Part 1
  Functional Programming (with Scala)

- Part 2
  Object-Oriented Programming (with Scala)

- Part 3
  Type Class (Type-Oriented) Programming (with Scala)

- Part 4
  Imperative Programming (with Rust)

Imperative vs. Functional Programming

- Imperative Programming
  - Computation by memory reads/writes
  - Sequence of read/write operations
  - Repetition by loop
  - More procedural (ie, describe how to do)
  - Easier to write efficient code

```
sum = 0;
i = n;
while (i > 0) {
  sum = sum + i;
  i = i - 1;
}
```

- Functional Programming
  - Computation by function application
  - Composition of function applications
  - Repetition by recursion
  - More declarative (ie, describe what to do)
  - Easier to write safe code

```
def sum(n: Int): Int
  if (n <= 0)
    0
  else
    n + sum(n-1)
```

Both Imperative & Functional Style Supported

- Many languages support both imperative & functional style
  - More imperative: Java, Javascript, C++, Python, Rust, …
  - More functional: OCaml, SML, Lisp, Scheme, …
  - Middle: Scala
  - Purely functional: Haskell (monadic programming)

Object-Oriented vs. Type-Oriented Programming

- Goal
  - To support module systems for writing large software
  - Specifically, to support code abstraction & reuse

- Object-Oriented Programming
  - Class Instance (ie, Object): data + methods (ie, functions)
  - Code abstraction & reuse together via inheritance

- Type Class (Type-Oriented) Programming
  - Type Class Instance: type + methods (ie, functions)
  - Abstraction and Reuse are separated
    - Code abstraction via type class instantiation
    - Code reuse via composition

Why Scala & Rust?

- Why Scala?
  - Equally well support both imperative & functional style
  - Many advanced features (both OOP & Type class supported)
  - Compatible with Java
  - Memory management with Garbage Collection (automatic, dynamic)

- Why Rust?
  - Can write (low-level) efficient code like C and C++ but safely
  - Using ownership types
  - Memory management with ownership types (automatic, static) + lightweight Garbage Collection

PART 1

Functional Programming with Function Applications

Values, Names, Functions and Evaluations

Values, Expressions, Names

- Types and Values
  - A type is a set of values
  - Int: {-2147483648,…,-1,0,1, …,2147483647} //32-bit integers
  - Double: 64-bit floating point numbers // real numbers in practice
  - Boolean: {true, false}
  - …
- Expressions
  - Composition of
    values, names, primitive operations, …
- Name Binding
  - Binding expressions to names
- Examples

```
def a = 1 + (2 + 3)
def b = 3 + a * 4
```

Evaluation

- Evaluation
  - Reducing an expression into a value
  - Strategy
  1. Take a name or an operator (outer to inner)
  2. (name) Replace the name with its associated expression
  3. (name) Evaluate the expression
  4. (operator) Evaluate its operands (left to right)
  5. (operator) Apply the operator to its operands

- Examples

```
5+b ~ 5+(3+(a*4)) ~ 5+(3+(1+(2+3))*4) ~ … ~ 32
```

Functions and Substitution

- Functions
  - Expressions with Parameters
  - Binding functions to names

```
def f(x: Int): Int = x + a
```

- Evaluation by substitution
  - …
  - (function) Evaluate its operands (left to right)
  - (function)
    Replace the function application by the expression of the function
    Replace its parameters with the operands

```
5+f(f(3)+1) ~ 5+f((3+a)+1) ~ … ~ 5+f(10) ~ 5+(10+a)
~ … ~ 21
```

Simple Recursion

- Recursion
  - Use X in the definition of X
  - Powerful mechanism for repetition
  - Nothing special but just rewriting

```
def sum(n: Int) : Int =
  if (n <= 0)
    0
  else
    n + sum(n-1)
```

```
sum(2) ~ if (2<=0) 0 else (2+sum(2-1)) ~
2+sum(1) ~ 2+(if (1<=0) 0 else (1+sum(1-1))) ~
2+(1+sum(0)) ~ 2+(1+(if (0<=0) 0 else (0+sum(0-1))))
~ 2+(1+0) ~ 3
```

Termination/Divergence

Evaluation may not terminate

- Termination
  - An expression may reduce to a value

- Divergence
  - An expression may reduce forever

```
def loop: Int = loop

loop ~ loop ~ loop ~ …
```

Evaluation strategy: Call-by-value, Call-by-name

```
f(e1,e2)
```

- Call-by-value
  - Evaluate the arguments first, then apply the function to them
- Call-by-name
  - Just apply the function to its arguments, without evaluating them.

```
def square (x: Int) = x * x

[cbv]square(1+1) ~ square(2) ~ 2*2 ~ 4

[cbn]square(1+1) ~ (1+1)*(1+1) ~ 2*(1+1) ~ 2*2 ~ 4
```

CBV, CBN: Differences

- Call-by-value
  - Evaluates arguments once
- Call-by-name
  - Do not evaluate unused arguments

- Question
  - Do both always result in the same value?

Scala's evaluation strategy

- Call-by-value
  - By default
- Call-by-name
  - Use "=>"

```
def one(x: Int, y: =>Int) = 1

one(1+2, loop)

one(loop, 1+2)
```

Scala's name binding strategy

- Call-by-value
  - Use "val" (also called "field") e.g. `val x = e`
  - Evaluate the expression first, then bind the name to it
- Call-by-name
  - Use "def" (also called "method") e.g. `def x = e`
  - Just bind the name to the expression, without evaluating it
  - Mostly used to define functions

```scala
def a = 1 + 2 + 3
val a = 1 + 2 + 3 // 6
def b = loop
val b = loop

def f(a: Int, b: Int): Int = a*b - 2
```

Conditional Expressions

- If-else
  - `if` (b) e_1 `else` e_2
  - b : Boolean expression
  - e_1, e_2: expressions of the same type
- Reduction rules:
  - `if (true)` e_1 `else` e_2 $\rightarrow$ e_1
  - `if (false)` e_1 `else` e_2 $\rightarrow$ e_2

```
def abs(x: Int) = if (x >= 0) x else -x
```

Boolean Expressions

- Boolean expression
  - `true, false`
  - `!b`
  - `b && b`
  - `b || b`
  - `e <= e, e >= e, e < e, e > e, e == e, e != e`
- Reduction rules:
  - `!true → false`
  - `!false → true`
  - `true && b → b`
  - `false && b → false`
  - `true || b → true`
  - `false || b → b`

`true && (loop == 1) ~ loop == 1 ~ loop == 1`

Exercise: and, or

- Write two functions
  - `and(x,y) == x && y`
  - `or(x,y) == x || y`
  - Do not use `&&`, `||`

```
and(false,loop==1)
~ if (false) loop==1 else false
~ false

and(true,loop==1)
~ if (true) loop==1 else false
~ loop==1 ~ loop==1 …
```

Solution

```scala
def and(x: Boolean, y: =>Boolean) =
  if (x) y else false

def or(x: Boolean, y: =>Boolean) =
  if (x) true else y
```

Exercise: square root calculation

➢Calculate square roots with Newton's method

```
def isGoodEnough(guess: Double, x: Double) =
  ??? // guess*guess is 99.9% close to x

def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2

def sqrtIter(guess: Double, x: Double): Double =
  ??? // repeat improving guess until it is good enough

def sqrt(x: Double) =
  sqrtIter(1, x)

sqrt(2)
```

Solution

➢Calculate square roots with Newton's method

```
def isGoodEnough(guess: Double, x: Double) =
  guess*guess/x > 0.999 && guess*guess/x < 1.001

def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2

def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess,x)) guess
  else sqrtIter(improve(guess,x),x)

def sqrt(x: Double) =
  sqrtIter(1, x)

sqrt(2)
```

Blocks and Name Scoping

Blocks in Scala

- Block
  - ```
    { val x1 = e1
      def x2 = e2
      e
    }
    ```
  - Is an expression
  - Allow nested name binding
  - Allow arbitrary order of "`def`"s, but not "`val`"s (think about why)
    ```

Scope of names

- Block

```
{ val t = 0
  def f(x: Int) = t + g(x+1)
  def g(y: Int) = y * y
  val x = f(5)
  val r = {
    val t = 10
    val s = f(5)
    s - t }
  t + r }
```

- Block-scoped definitions are only accessible within the block
- Inner definitions shadow outer ones of the same name
- Outer definitions are accessible in nested blocks unless shadowed
- No duplicate definitions allowed in the same block
- Functions are evaluated under the environment where they are defined, not where they are called

Problem

```
// should be allowed
{
  def f(x:Int) = g(x)
  def g(x: Int) = 10
  val x = f(10)
  x
}

// should not be allowed
{
  def f(x:Int) = g(x)
  val x = f(10)
  def g(x: Int) = 10
  x
}
```

Safety Checking

Safety checking rules out any use of undefined names.

- For “val x = e”,
  all names in “e” should be defined before this definition.
- For “def x = e”,
  all names in “e” should be defined before the next “val” definition.

/* The following code passes the safety checking */

```
{ def f(x:Int) = g(x)
  def g(x: Int) = 10
  val a = 10
  f(10) }
```

/* The following code fails at the safety checking */

```
{ def f(x:Int) = g(x)
  val a = 10
  def g(x: Int) = 10
  f(10) }
```

Evaluation for Blocks

```
1: { val t = 0
2:   def f(x: Int) = t + g(x+1)
3:   def g(y: Int) = y*y
4:   val x = f(5) + 7
5: val r = {
6:     val t = 10
7:     val s = f(5)
8:     s - t }
9: t + r }
```

- Evaluation with Environment

```
[E0|t=_,f=_,g=_,x=_ ,r=_ ],1 ~ 0
[E0|t=0,f=(x)t+g(x),g=(y)y*y,x=_ ,r=_ ],4 ~ f(5)+7 ~ 36+7~ 43
  f(5): E0::[E1|x=5],t+g(x+1) ~ 0+g(6) ~ 36
          g(6): E0::[E2|y=6],y*y ~ 6*6 ~ 36
[E0|t=0,f=(x)t+g(x),g=(y)y*y,x=43,r=_ ],5 ~ 26
  5: E0::[E3|t=_ ,s=_ ],6 ~ 10
     E0::[E3|t=10,s=_ ],7 ~ f(5) ~ 36
       f(5): E0::[E4|x=5],t+g(x+1) ~ 0+g(6) ~ 36
               g(6): E0::[E5|y=6],y*y ~ 6*6 ~ 36
     E0::[E3|t=10,s=36],8 ~ s-t ~ 26
[E0|t=0,f=(x)t+g(x),g=(y)y*y,x=43,r=26],9 ~ t+r ~ 26
```

Semi-colons and Parenthesis

- Block
  - Can write two definitions/expressions in a single line using ;
  - Can write one definition/expression in two lines using (), but can omit () when clear

```
// ok
val r = {
  val t = 10; val s = square(5); t +
  s }
// Not ok
val r = {
  val t = 10; val s = square(5); t
  + s }
// ok
val r = {
  val t = 10; val s = square(5); (t
  + s) }
```

Exercise: Writing Better Code using Blocks

- Make the following code better

```
def isGoodEnough(guess: Double, x: Double) =
  guess*guess/x > 0.999 && guess*guess/x < 1.001

def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2

def sqrtIter(guess: Double, x: Double): Double = {
  if (isGoodEnough(guess,x)) guess
  else sqrtIter(improve(guess,x),x)
}

def sqrt(x: Double) =
  sqrtIter(1, x)

sqrt(2)
```

Solution

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double = {
    if (isGoodEnough(guess,x)) guess
    else sqrtIter(improve(guess,x),x)
  }
  def isGoodEnough(guess: Double, x: Double) = {
    val ratio = guess * guess / x
    ratio > 0.999 && ratio < 1.001
  }
  def improve(guess: Double, x: Double) =
    (guess + x/guess) / 2

  sqrtIter(1, x)
}

sqrt(2)
```

Lazy Call-By-Value

Lazy call-by-value

- Lazy call-by-value
  - Use "lazy val" e.g. `lazy val x = e`
  - Evaluate the expression first time it is used, then bind the name to it

```
def f(c: Boolean, i: =>Int): Int = {
  lazy val iv = i
  if (c) 0
  else   iv * iv * iv
}

f(true, {println("ok"); 100+100+100+100})
f(false, {println("ok"); 100+100+100+100})
```

Tail Recursion & Tail Call

Recursion needs care

- Summation function
  - Write a summation function `sum` such that
    `sum(n) = 1+2+…+n`
  - Test
    `sum(10),sum(100),sum(1000),sum(10000),`
    `sum(100000), sum(1000000)`
  - What's wrong? (Think about evaluation)

Recursion: Try

```
def sum(n: Int): Int =
  if (n <= 0) 0 else (n+sum(n-1))
```

Recursion: Tail Recursion

```
import scala.annotation.tailrec

def sum(n: Int): Int = {
  @tailrec def sumItr(res: Int, m: Int): Int =
    if (m <= 0) res else sumItr(m+res,m-1)
  sumItr(0,n)
}
```

Mutual Recursion: Try

```
{
  def sum(acc: Int, n: Int): Int =
    if (n <= 0) acc else sum2(n + acc, n-1)

  def sum2(acc: Int, n: Int): Int =
    if (n <= 0) acc else sum(2*n + acc, n-1)

  sum(0, 20000)  // stack overflow
}
```

Mutual Recursion: Tail Call Optimization

```scala
import scala.util.control.TailCalls._

{
  def sum(acc: Int, n: Int): TailRec[Int] =
    if (n <= 0) done(acc) else tailcall(sum2(n + acc, n-1))

  def sum2(acc: Int, n: Int): TailRec[Int] =
    if (n <= 0) done(acc) else tailcall(sum(2*n + acc, n-1))

  sum(0, 20000).result
}
```

Higher-Order Functions

Functions as Values

- Functions
  - Functions are normal values of function types ($A_1, \ldots, A_n$=>B).
  - They can be copied, passed and returned.
  - Functions that take functions as arguments or return functions are called higher-order functions.
  - Higher-order functions increase code reusability.

Examples

```
def sumLinear(n: Int): Int =
  if (n <= 0) 0 else n + sumLinear(n-1)

def sumSquare(n: Int): Int =
  if (n <= 0) 0 else n*n + sumSquare(n-1)

def sumCubes(n: Int): Int =
  if (n <= 0) 0 else n*n*n + sumCubes(n-1)
```

Q: How to write reusable code?

Examples

```
def sum(f: Int=>Int, n: Int): Int =
  if (n <= 0) 0 else f(n) + sum(f, n-1)

def linear(n: Int) = n
def square(n: Int) = n * n
def cube(n: Int) = n * n * n

def sumLinear(n: Int) = sum(linear, n)
def sumSquare(n: Int) = sum(square, n)
def sumCubes(n: Int) = sum(cube, n)
```

Anonymous Functions

- Anonymous Functions
  - Syntax

    $(x_1: T_1,\ldots,x_n:T_n) \Rightarrow e$

    or

    $(x_1,\ldots,x_n) \Rightarrow e$

```
def sumLinear(n: Int) = sum((x:Int)=>x, n)
def sumSquare(n: Int) = sum((x:Int)=>x*x, n)
def sumCubes(n: Int) = sum((x:Int)=>x*x*x, n)
```

Or simply

```
def sumLinear(n: Int) = sum((x)=>x, n)
def sumSquare(n: Int) = sum((x)=>x*x, n)
def sumCubes(n: Int) = sum((x)=>x*x*x, n)
```

Exercise

```
def sum(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f, a+1, b) else 0

def product(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) * product(f, a+1, b) else 1
```

DRY (Do not Repeat Yourself) using a higher-order function, called "mapReduce".

Exercise

```
def mapReduce(reduce: (Int,Int)=>Int,inival: Int,
              f: Int=>Int, a: Int, b: Int): Int = {
  if (a <= b) reduce(f(a),mapReduce(reduce,inival,f,a+1,b))
  else inival
}

def sum(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x+y,0,f,a,b)

def product(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x*y,1,f,a,b)
```

Evaluation for functional values

```
1: { val t = 0
2:     val f : Int=>Int = {      // Note: What if using "def f" ?
3:         val t = 10
4:         def g(x: Int) : Int = x + t
5:         g _ }
6:     f(20) }
```

* Try: Evaluation without Closures

```
[E0|t=_,f=_],1 ~ 0
[E0|t=0,f=_],2 ~ (x)x+t
  E0::[E1|t=_,g=(x)x+t],3 ~ 10
  E0::[E1|t=10,g=(x)x+t],5 ~ (x)x+t
[E0|t=0,f=(x)x+t],6 ~ f(20) ~ 20
  f(20): E0::[E2|x=20],x+t ~ 20+0 ~ 20
```

Closures for functional values

```
1: { val t = 0
2:     val f : Int=>Int = {      // Note: What if using "def  f" ?
3:         val t = 10
4:         def g(x: Int) : Int = x + t
5:         g _ }
6:     f(20) }
```

* Evaluation with Closures

```
[E0|t=_,f=_],1 ~ 0
[E0|t=0,f=_],2 ~ (E1,(x)x+t)
  E0::[E1|t=_,g=(x)x+t],3 ~ 10
  E0::[E1|t=10,g=(x)x+t],5 ~ (E1,(x)x+t)
[E0|t=0,f=(E1,(x)x+t)],6 ~ f(20) ~ 30
  f(20): E1::[E2|x=20],x+t ~ 20+10 ~ 30
```

(Parameterized) expression vs. (closure) value

- Functions defined using "def" are not values but parameterized expressions.
- Parameterized expression `f` can be converted to a value `(f _)`.
- The compiler often infers and inserts missing conversions automatically.
- Anonymous functions are values.
- Anonymous functions can be seen as syntactic sugar:

  ```
  (x:T)=>e
  ```

  is equivalent to

  ```
  { def noname(x:T) = e; (noname _) }
  ```

  where `noname` is not used in `e`.
- One can even write a recursive anonymous function in this way.
- Q: what's the difference between param. exps and function values?
  A: function values are "closures" (ie, param. exp. + env.)
- Q: how to implement call-by-name?
  A: The argument expression is converted to a closure.

Example: call by name with closures

```
1: { val t = 0
2:     def f(x: =>Int) = t + x
3:     val r = {
4:       val t = 10
5:       f(t*t) }                  // t*t is treated as ()=>t*t
6:     r }
```

➢Evaluation with Closures

```
[E0|t=_,f=(x:=>Int)t+x,r=_],1 ~ 0
[E0|t=0,f=(x:=>Int)t+x,r=_],3 ~ 100
  E0::[E1|t=_],4 ~ 10
  E0::[E1|t=10],5 ~ f(t*t) ~ 100
  f(t*t): E0::[E2|x=(E1,t*t)],t+x ~ 0+x ~ 0+100 ~ 100
    x: E1,t*t ~ 10*10 ~ 100
[E0|t=0,f=(x)t+x,r=100],6 ~ r ~ 100
```

Currying

Motivation

```
def sum(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f, a+1, b) else 0
def sumLinear(a: Int, b: Int) = sum((n)=>n, a, b)
def sumSquare(a: Int, b: Int) = sum((n)=>n*n, a, b)
def sumCubes(a: Int, b: Int) = sum((n)=>n*n*n, a, b)
```

We want the following. How?

```
val sumLinear = sum(linear)
val sumSquare = sum(square)
val sumCubes = sum(cube)
```

Solution

```
def sum(f: Int=>Int): (Int,Int)=>Int = {
  def sumF(a: Int, b: Int): Int =
    if (a <= b) f(a) + sumF(a+1, b) else 0
  sumF  // sumF _
}

def sumLinear = sum((n)=>n)
def sumSquare = sum((n)=>n*n)
def sumCubes = sum((n)=>n*n*n)
```

Benefits

```
def sumLinear = sum((n)=>n)
def sumSquare = sum((n)=>n*n)
def sumCubes = sum((n)=>n*n*n)

sumSquare(3,10) + sumCubes(5,20)
```

We don't need to define the wrapper functions.

```
sum((n)=>n*n)(3,10) + sum((n)=>n*n*n)(5,20)
```

Multiple Parameter List

```
def sum(f: Int=>Int): (Int,Int)=>Int = {
  def sumF(a: Int, b: Int): Int =
    if (a <= b) f(a) + sumF(a+1, b) else 0
  sumF _
}
```

Or simply:

```
def sum(f: Int=>Int)(a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f)(a+1, b) else 0
```

Note that `sum(f)` is just a parameterized expression.
`(sum(f) _)` creates a closure like `(sumF _)`

The following code is slightly inefficient. Think about why.

```
def sum(f: Int=>Int): (Int,Int)=>Int =
  (a,b) => if (a <= b) f(a) + sum(f)(a+1, b) else 0
```

Comparison between Param. Exp vs. Closure

```
{
  def sum(f: Int=>Int)(a: Int, b: Int): Int =
    if (a <= b) f(a) + sum(f)(a+1, b) else 0
  def sumLinear = sum((n)=>n) _ // sum((n)=>n) incorrect

  sumLinear(0,100)
}
{
  def sum(f: Int => Int): (Int, Int) => Int = {
    def sumF(a: Int, b: Int): Int =
      if (a <= b) f(a) + sumF(a + 1, b) else 0
    sumF _
  }
  def sumLinear = sum((n)=>n) // sum((n)=>n) _ incorrect

  sumLinear(0,100)
}
```

Currying and Uncurrying

- A function of type

$$(T_1,T_2,\ldots,T_n)\Rightarrow T$$

can be turned into that of type

$$T_1\Rightarrow(T_2\Rightarrow(\ldots\Rightarrow(T_n\Rightarrow T)\ldots))$$

- This is called "currying" named after Haskell Brooks Curry.
- The opposite direction is called "uncurrying".

Currying using Anonymous Functions

```
def foo(x: Int, y: Int, z: Int)(a: Int, b: Int) =
  x + y + z + a + b

val f1 = (x: Int, z: Int, b: Int) => foo(x,1,z)(2,b)

val f2 = foo(_:Int,1,_:Int)(2, _:Int)

val f3 = (x: Int, z: Int) => ((b: Int) => foo(x,1,z)(2,b))

f1(1,2,3)
f2(1,2,3)
f3(1,2)(3)
```

Exercise

Curry the mapReduce function.

```
def mapReduce(reduce: (Int,Int)=>Int,inival: Int,
              f: Int=>Int, a: Int, b: Int): Int = {
  if (a <= b) reduce(f(a),mapReduce(reduce,inival,f,a+1,b))
  else inival
}

def sum(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x+y,0,f,a,b)

def product(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x*y,1,f,a,b)
```

Solution

```
def mapReduce(reduce:(Int,Int)=>Int,inival: Int)
             (f: Int=>Int) (a: Int, b: Int): Int = {
  if (a <= b) reduce(f(a),mapReduce(reduce,inival)(f)(a+1,b))
  else inival
}

// need to make a closure since mapReduce is param. code.
def sum = mapReduce((x,y)=>x+y,0) _

// val is better than def. Think about why.
val product = mapReduce((x,y)=>x*y,1) _
```

Exceptions

Exception & Handling

```
class factRangeException(val arg: Int) extends Exception

def fact(n : Int): Int =
  if (n < 0) throw new factRangeException(n)
  else if (n == 0) 1
  else n * fact(n-1)

def foo(n: Int) = fact(n + 10)

try {
  println (fact(3))
  println (foo(-100))
} catch {
  case e : factRangeException => {
    println("fact range error: " + e.arg)
  }
}
```

Datatypes

Types so far

Types have introduction operations and elimination ones.

- Introduction: how to construct elements of the type
- Elimination: how to use elements of the type

- Primitive types
  - Int, Boolean, Double, String, …
  - Intro for Int: …,-2,-1,0,1,2,
  - Elim for Int: +,-,*,/,<,<=,…

- Function types
  - Int=>Int, (Int=>Int)=>(Int=>Int), …
  - Intro: (x:T)=>e
  - Elim: f(v)

Tuples

- Tuples

  Intro:

  - (1,2,3) : (Int, Int, Int)
  - (1,“a”) : (Int, String)

  Elim:

  - (1, “a”,10)._1 = 1
  - (1, “a”,10)._2 = “a”
  - (1, “a”,10)._3 = 10

  Only up to length 22

Structural Types (a.k.a. Record Types): Examples

```
import reflect.Selectable.reflectiveSelectable
def bar (x: Int) = x+1

val foo = new {
  val a = 1+2
  def b = a + 1
  def f(x: Int) = b + x
  val g : Int => Int = bar _
}

foo.b

foo.f(3)

val ff : Int=>Int = foo.f _

def g(x: {val a: Int; def b: Int;
          def f(x:Int): Int; val g: Int => Int}) =
  x.f(3)
g(foo)
```

Type Alias

```
import reflect.Selectable.reflectiveSelectable
type Foo = {val a: Int; def b: Int; def f(x:Int):Int}

val gn = 0
val foo : Foo = new {
  val a = 3
  def b = a + 1
  def f(x: Int) = b + x + gn
}

foo.f(3)

def g(x: Foo) = {
  val gn = 10
  x.f(3)
}
g(foo)
```

Structural Types: Evaluation

```
1: def bar (x: Int) = x+1
2: val foo = new //or, object foo
3: { val a = 2 + 1
4:   def f(x: Int) = a + x
5:   val g : Int => Int = bar _
6: }
7: val b = foo.f(1)
8: foo.g(b)
```

- Evaluation with Closures

```
E1[],1 ~ E1[bar=(x)x+1],2 ~ E1[…]:E2[],3 ~
E1[…]:E2[a=3],4 ~
E1[…]:E2[a=3,f=(x)a+x],5 ~
E1[…]:E2[a=3,f=(x)a+x,g=((x)x+1,E1)],6 ~
E1[bar=(x)x+1,foo=(E2)],7 ~
E1[bar=(x)x+1,foo=(E2),b=4],8 ~ 5
7: E1:E2:E3[x=1],a+x ~ 3+1 ~ 4
8: E1:E4[x=4],x+1 ~ 4+1 ~ 5
```

Algebraic Datatypes

- Ideas
  - T = C of T * … * T
    | C of T * … * T
    | …
    | C of T * … * T

  - E.g.
    Attr = Name of String
    | Age of Int
    | DOB of Int * Int * Int
    | Height of Double
    Intro:
    Name("Chulsoo Kim"), Name("Younghee Lee"), Age(16), DOB(2000,3,10), Height(171.5), …

Algebraic Datatypes: Recursion

- Recursive ADT
  - E.g.

    IList = INil

    | ICons of Int * IList

    Intro:

    INil(),

    ICons(3, INil()),

    ICons(2, ICons(3, INil())),

    ICons(1, ICons(2, ICons(3, INil()))),

    …

Algebraic Datatypes In Scala

- Attr

```scala
sealed abstract class Attr
case class Name(name: String) extends Attr
case class Age(age: Int) extends Attr
case class DOB(year: Int, month: Int, day: Int) extends Attr
case class Height(height: Double) extends Attr

val a : Attr = Name("Chulsoo Kim")
val b : Attr = DOB(2000,3,10)
```

- IList

```scala
sealed abstract class IList
case class INil() extends IList
case class ICons(hd: Int, tl: IList) extends IList

val x : IList = ICons(2, ICons(1, INil()))
def gen(n: Int) : IList =
  if (n <= 0) INil()
  else ICons(n, gen(n-1))
```

Exercise

```
IOption = INone
        | ISome of Int
```

```
BTree = Leaf
      | Node of Int * BTree * BTree
```

```scala
sealed abstract class IList
case class INil() extends IList
case class ICons(hd: Int, tl: IList) extends IList

def x : IList = ICons(2, ICons(1, INil()))
```

Solution

```
sealed abstract class IOption
case class INone() extends IOption
case class ISome(some: Int) extends IOption

sealed abstract class BTree
case class Leaf() extends BTree
case class Node(value: Int, left: BTree, right: BTree)
  extends BTree
```

Pattern Matching

- Pattern Matching
  - A way to use algebraic datatypes

```
e match {
  case C1(…) => e1 : T
   …
  case Cn(…) => en : T
} : T
```

Pattern Matching: An Example

```
def length(xs: IList) : Int =
  xs match {
    case INil() => 0
    case ICons(x, tl) => 1 + length(tl)
  }

length(x)
```

Advanced Pattern Matching

- Advanced Pattern Matching

```
e match {
  case P1 => e1
   …
  case Pn => en
}
```

  - One can combine constructors and use _ and | in a pattern.
    (E.g) case ICons(x, INil()) | ICons(x, ICons(_, INil())) => …
  - The given value `e` is matched against the first pattern `P1`.
    If succeeds, evaluate `e1`.
    If fails, `e` is matched against `P2`.
    If succeeds, evaluate `e2`.
    If fails, …
  - The compiler checks exhaustiveness (ie, whether there is a missing case).

Advanced Pattern Matching: An Example

```
def secondElmt(xs: IList) : IOption =
  xs match {
    case INil() | ICons(_,INil()) => INone()
    case ICons(_, ICons(x, _)) => ISome(x)
  }
```

Vs.

```
def secondElmt2(xs: IList) : IOption =
  xs match {
    case INil() | ICons(_,INil()) => INone()
    case ICons(_, ICons(x, INil())) => ISome(x)
    case _ => INone()
  }
```

Vs.

```
def secondElmt2(xs: IList) : IOption =
  xs match {
    case ICons(_, ICons(x, INil())) => ISome(x)
    case _ => INone()  }
```

Pattern Matching on Int

```
def factorial(n: Int) : Int =
  n match {
    case 0 => 1
    case _ => n * factorial(n-1)
  }

def fib(n: Int) : Int =
  n match {
    case 0 | 1 => 1
    case _ => fib(n-1) + fib(n-2)
  }
```

Pattern Matching with If

```
def f(n: Int) : Int =
  n match {
    case 0 | 1 => 1
    case _ if (n <= 5) => 2
    case _ => 3
  }

def f(t: BTree) : Int =
  t match {
    case Leaf() => 0
    case Node(n,_,_) if (n <= 10) => 1
    case Node(_,_,_) => 2
  }
```

Exercise

Write a function “find(t: BTree, x: Int) : Boolean” that checks whether x is in t.

```
sealed abstract class BTree
case class Leaf() extends BTree
case class Node(value: Int, left: BTree, right: BTree)
  extends BTree
```

Solution

```
def find(t: BTree, i: Int) : Boolean =
  t match {
    case Leaf() => false
    case Node(n,lt,rt) =>
      i == n || find(lt, i) || find(rt, i)
  }

def t: BTree = Node(5,Node(4,Node(2,Leaf(),Leaf()),Leaf()),
  Node(7,Node(6,Leaf(),Leaf()),Leaf()))
find(t,7),find(t,1)
```

Type Checking & Inference (Concept)

What Are Types For?

- Typed Programming

```
def id1(x: Int): Int = x
def id2(x: Double): Double = x
```

  - At run time, type information is erased (ie, `id1` = `id2`)
- Untyped Programming

```
def id(x) = x
```

  - Do not care about types at compile time.
  - But, many such languages check types at run time paying cost.
  - Without run-time type check, errors can be badly propagated.
- What is compile-time type checking for?
  - Can detect type errors at compile time.
  - Increase Readability (Give a good abstraction).
  - Soundness: Well-typed programs raise no type errors at run time.

Type Checking and Inference

- Type Checking

$$x_1{:}T_1,\ x_2{:}T_2,\ \ldots,\ x_n{:}T_n \ \vdash\ e : T$$

  - `def f(x: Boolean): Boolean = x > 3`

    => Type error
  - `def f(x: Int): Boolean = x > 3`

    => OK. `f: (x: Int)Boolean`

- Type Inference

$$x_1{:}T_1,\ x_2{:}T_2,\ \ldots,\ x_n{:}T_n \ \vdash\ e : ?$$

  - `def f(x: Int) = x > 3`

    => OK by type inference. `f: (x: Int)Boolean`
  - Too much type inference is not good. Why?

You can learn how type checking & inference work in
4190.310 Programming Languages

Parametric Polymorphism

Parametric Polymorphism: Functions

- Problem

```
def id1(x: Int): Int = x
def id2(x: Double): Double = x
```

  - Can we avoid copy and paste?
  - Polymorphism to the rescue!

- Parametric Polymorphism (a.k.a. For-all Types)

```
def id[A](x: A) : A = x
```

  - The type of `id` is `[A](val x:A)=>A`
  - `id` is a parametric expression.
  - `id[T] _` is a value of type `T=>T` for any type `T`.
  - Function types do not support polymorphism.
    (E.g.) `[A](A=>A)` is not a valid function value type.

[We will learn other kinds of polymorphism later.]

Examples

```
def id[A](x:A) = x
id(3)
id("abc")

def applyn[A](f: A => A, n: Int, x: A): A =
  n match {
    case 0 => x
    case _ => f(applyn(f, n - 1, x))
  }
applyn((x:Int)=>x+1,100,3)
applyn((x:String)=>x+"!", 10, "gil")
applyn(id[String], 10, "hur")

def foo[A,B](f: A=>A, x: (A,B)) : (A,B) =
  (applyn[A](f, 10, x._1), x._2)

foo[String,Int]((x:String)=>x+"!",("abc",10))
```

Parametric Polymorphism: Datatypes

```
sealed abstract class MyOption[A]
case class MyNone[A]() extends MyOption[A]
case class MySome[A](some: A) extends MyOption[A]

sealed abstract class MyList[A]
case class MyNil[A]() extends MyList[A]
case class MyCons[A](hd: A, tl: MyList[A]) extends MyList[A]

sealed abstract class BTree[A]
case class Leaf[A]() extends BTree[A]
case class Node[A](value: A, left: BTree[A], right: BTree[A])
extends BTree[A]

def x: MyList[Int] = MyCons(3, MyNil())
def y: MyList[String] = MyCons("abc", MyNil())
```

Revisit: Solution with Tail Recursion

```
def find[A](t: BTree[A], x: A) : Boolean = {
  def findIter(ts: MyList[BTree[A]]) : Boolean =
    ts match {
      case MyNil() => false
      case MyCons(Leaf(), tl) => findIter(tl)
      case MyCons(Node(v, _, _), _) if v == x => true
      case MyCons(Node(_,l,r), tl) =>
        findIter(MyCons(l, MyCons(r, tl)))    }
  findIter(MyCons(t, MyNil()))
}

def genTree(v: Int, n: Int) : BTree[Int] = {
  def genTreeIter(t: BTree[Int], m : Int) : BTree[Int] =
    if (m == 0) t
    else genTreeIter(Node(v, t, Leaf()), m-1)
  genTreeIter(Leaf(), n)
}

find(genTree(0,10000), 1)
```

Exercise

BSTree[A] = Leaf
| Node of Int * A * BSTree[A] * BSTree[A]

```
def lookup[A](t: BSTree[A], k: Int) : MyOption[A] =
  ???

def t : BSTree[String] =
  Node(5,"My5",
    Node(4,"My4",Node(2,"My2",Leaf(),Leaf()),Leaf()),
    Node(7,"My7",Node(6,"My6",Leaf(),Leaf()),Leaf()))

lookup(t, 7)
lookup(t, 3)
```

Solution

```scala
sealed abstract class BSTree[A]
case class Leaf[A]() extends BSTree[A]
case class Node[A](key: Int, value: A, left: BSTree[A], right:
BSTree[A]) extends BSTree[A]
def lookup[A](t: BSTree[A], key: Int) : MyOption[A] =
  t match {
    case Leaf() => MyNone()
    case Node(k,v,lt,rt) =>
      k match {
        case _ if key == k => MySome(v)
        case _ if key < k => lookup(lt,key)
        case _ => lookup(rt, key)
      }
  }
def t : BSTree[String] =
  Node(5,"My5",
    Node(4,"My4",Node(2,"My2",Leaf(),Leaf()),Leaf()),
    Node(7,"My7",Node(6,"My6",Leaf(),Leaf()),Leaf()))
lookup(t, 7)
lookup(t, 3)
```

A Better Way

```
sealed abstract class BTree[A]
case class Leaf[A]() extends BTree[A]
case class Node[A](value: A, left: BTree[A], right: BTree[A])
  extends BTree[A]

type BSTree[A] = BTree[(Int,A)]

def lookup[A](t: BSTree[A], k: Int) : MyOption[A] =
  ???

def t : BSTree[String] =
  Node((5,"My5"),
    Node((4,"My4"),Node((2,"My2"),Leaf(),Leaf()),Leaf()),
    Node((7,"My7"),Node((6,"My6"),Leaf(),Leaf()),Leaf()))

lookup(t, 7)
```

Solution

```scala
type BSTree[A] = BTree[(Int,A)]

def lookup[A](t: BSTree[A], key: Int) : MyOption[A] =
  t match {
    case Leaf() => MyNone()
    case Node((k,v),lt,rt) =>
      k match {
        case _ if key == k => MySome(v)
        case _ if key < k => lookup(lt,key)
        case _ => lookup(rt, key)
      }
  }

def t : BSTree[String] =
  Node((5,"My5"),
    Node((4,"My4"),Node((2,"My2"),Leaf(),Leaf()),Leaf()),
    Node((7,"My7"),Node((6,"My6"),Leaf(),Leaf()),Leaf()))

lookup(t, 7)
lookup(t, 3)
```

Polymorphic Option (Library)

- Option[T]

  Intro:
  - None
  - Some(x)
  - Library functions

  Elim:
  - Pattern matching
  - Library functions

  Some(3) : Option[Int]
  Some("abc"): Option[String]
  None: Option[Int]
  None: Option[String]

Polymorphic List (Library)

- List[T]

  Intro:
  - Nil
  - x :: L
  - Library functions

  Elim:
  - Pattern matching
  - Library functions

  "abc"::Nil : List[String]

  List(1,3,4,2,5) = 1::3::4::2::5::Nil : List[Int]

Summary: Parametric Polymorphism

- Parametric Polymorphism
  - Program against unknown datatypes
  - How is it possible?