# Principles of (Functional) Programming

(4190.306)

Chung-Kil Hur (허충길)

Department of Computer Science and Engineering
Seoul National University

# Syllabus

- Lecture
  - Mon & Tue, 9:00 ~ 10:50 (302-208)
  - https://github.com/snu-sf-class/pp201602
- Instructor
  - Chung-Kil Hur
  - http://sf.snu.ac.kr/gil.hur/
- Teaching Assistant
  - Youngju Song
  - http://sf.snu.ac.kr/youngju.song/
- Grading
  - Attendance: 5%
  - Assignments: 25%
  - Midterm exam: 30%
  - Final exam: 40%

# Introduction

# Imperative vs. Functional Programming

- Imperative Programming
  - Computation by memory reads/writes
  - Sequence of read/write operations
  - Repetition by loop
  - More procedural
  - 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
  - Easier to write safe code

```
def sum(n) =
  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, …
  - More functional: OCaml, SML, Lisp, Scheme, …
  - Middle: Scala
  - Purely functional: Haskell

- Why Scala?
  - Equally well support both imperative & functional style
  - A lot of advanced features
  - Compatible with Java

# 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 (= Programming)
  - 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) ~ … ~ 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) =
  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 binding 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

```
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
- Rewrite 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`
- Rewrite 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 …
```

# Exercise: square root calculation

- Calculate square roots with Newton's method

```
def isGoodEnough(guess: Double, x: Double) =
  ??? // guess*guess is 99% 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)
```

# 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 square(x: Int) = t + x * x
val x = square(5)
val r = {
  val t = 10
  val s = square(5)
  t + s
}
val y = t + r
```

  - A definition inside a block is only accessible within the block
  - A definition inside a block shadows definitions of the same name outside the block
  - A definition inside a block is accessible unless it is shadowed
  - A function is evaluated under the environment where it is defined, not the environment where it is invoked.

# Rewriting for blocks

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

➢Evaluation by rewriting

[f=(x)t+g(x),g=(x)x*x],1 ~ […,t=0],2 ~ […],3 ~ […],4 ~ […,x=25],5 ~ […]:[],6 ~ […]:[t=10],7 ~ […]:[…,s=25],8 ~ […,r=35],9 ~ […,y=35],10

4: [f=…,g=…,t=0]:[x=5],t+g(x) ~…~ […]:[…],25

7: [f=…,g=…,t=0]:[x=5],t+g(x) ~…~ […]:[…],25

# 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)
```

# Recursion

# 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 1

```
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)
}
```

# Higher-Order Functions

# Functions as Values

- Functions
  - Functions are normal values of function types $(A_1,\ldots,A_n\Rightarrow B)$.
  - 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

```scala
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(combine:(Int,Int)=>Int,inival: Int,
              f: Int=>Int, a: Int, b: Int): Int = {
  if (a <= b) combine(f(a),mapReduce(combine,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)
```

# Parameterized expression vs. values

- Functions defined using "def" are not values but parameterized expressions.
- Anonymous functions are values.
- But, parameterized expressions are implicitly converted to values.
- Explicit conversion: f _
- Anonymous functions can be seen as syntactic sugar:
```
(x:T)=>e
```
  is equivalent to
```
{ def __noname(x:T)=>e; __noname _ }
```
- One can even write a recursive anonymous function in this way.

- Q: what's the difference between param. exps and function values?
  A: functions values are "closures" (ie, param. exp. + env.)
- Q: how to implement call-by-name?
  A: The argument expression is converted to a closure.

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

We want the following. How?

```
def sumLinear = sum(linear)
def sumSquare = sum(square)
def 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
}

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

# Benefits

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

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

We don't need to define the wrapper functions.

```
sum(square)(3,10) + sum(cube)(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
}
```

We can also write as follows.

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

Or more simply:

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

# Currying and Uncurrying

- A function of type

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

can be turned into one of type

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

- 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.

# Solution

```scala
def mapReduce(combine:(Int,Int)=>Int,inival: Int)
             (f: Int=>Int) (a: Int, b: Int): Int = {
  if (a <= b) combine(f(a),mapReduce(combine,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) _
```

# 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

# Record Types: Examples

```
object foo {
  val a = 3
  def b = a + 1
  def f(x: Int) = b + x
}

foo.f(3)

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

g(foo)
```

# Record Types: Scope and Type Alias

```scala
val gn = 0
object foo {
  val a = 3
  def b = a + 1
  def f(x: Int) = b + x + gn
}

foo.f(3)

type Foo = {val a: Int; def b: Int; def f(x:Int):Int}

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

g(foo)
```

# 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(1,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

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

- IList

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

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

# Exercise

IOption = INone
  | ISome of Int

BTree = Leaf
  | Node of Int * BTree * BTree

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

sealed abstract class BTree
case object 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
   …
  case Cn(…) => en
}
```

# 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.

# 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
  }
```

# 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) that checks whether x is in t.

# Solution

```scala
def find(t: BTree, i: Int) : Boolean =
  t match {
    case Leaf => false
    case Node(n,lt,rt) =>
      if (i == n) true
      else if (i < n) find(lt, i)
      else 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)
```

# 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.

- Why 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 run 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 DRY?
  - Polymorphism to the rescue!

- Parametric Polymorphism

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

  - The type of `id` is `[A](x:A)A`
  - `id` is a parametric expression.
  - `id[T] _` is a value of type `T=>T` for any type `T` .

[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))
```

# Full Polymophism using Scala's trick

```scala
type Applyn = {def apply[A](f: A=>A, n: Int, x: A): A}

object applyn {
  def apply[A](f: A=>A, n: Int, x: A): A =
    n match {
      case 0 => x
      case _ => f(apply(f, n-1, x))
    }
}

applyn((x:String)=>x+"!", 10, "gil")

def foo(f: Applyn): String = {
  val a:String = f[String]((x:String)=> x + "!",10,"gil")
  val b:Int = f[Int]((x:Int)=> x + 2, 10, 5)

  a + b.toString()
}
```

# Parametric Polymorphism: Datatypes

```
sealed abstract class MyOption[+A]
case object MyNone extends MyOption[Nothing]
case class MySome[A](some: A) extends MyOption[A]

sealed abstract class MyList[+A]
case object MyNil extends MyList[Nothing]
case class MyCons[A](hd: A, tl: MyList[A]) extends MyList[A]

sealed abstract class BTree[+A]
case object Leaf extends BTree[Nothing]
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)
```

# 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

```
sealed abstract class BSTree[+A]
case object Leaf extends BSTree[Nothing]
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 object Leaf extends BTree[Nothing]
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

```
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)
```

# Parametric Polymorphism: Datatypes, Generally

MyType[A] = MyNone
| MyFun of A=>Boolean

```
sealed abstract class MyType[+A]
case object MyNone extends MyType[Nothing]
case class MyFun[A](fun: A=>Boolean) extends MyType[A]

def foo[A](d:MyType[A], x:A): Boolean =
  d match {
    case MyFun(f) => f(x)
    case _ => false
  }
```

Q: What's wrong here?

A: It is not monotone.

# Parametric Polymorphism: Datatypes, Generally

```
sealed abstract class MyType[+A]
case object MyNone extends MyType[Nothing]
case class MyFun[A](fun: A=>Boolean) extends MyType[A]

->

sealed abstract class MyType[A]
case class MyNone[A]() extends MyType[A]
case class MyFun[A](fun: A=>Boolean) extends MyType[A]
```

- Cannot use “case object”: no support of parametric record types
- Have to use () for MyNone (only minor downside)

# 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]

# Sub Type Polymorphism (Concept)

# Motivation

We want:

```
object tom {
  val name = "Tom"
  val home = "02-880-1234"
}
object bob {
  val name = "Bob"
  val mobile = "010-1111-2222"
}
```

```
def greeting(r: ???) = "Hi " + r.name + ", How are you?"
greeting(tom)
```

```
greeting(bob)
```

We Note that we have

```
tom: {val name: String; val home: String}
bob: {val name: String; val mobile: String}
```

# Sub Types to the Rescue!

```
type NameHome = { val name: String; val home: String }
type NameMobile = { val name: String; val mobile: String}
type Name = { val name: String }
```

`NameHome <: Name` (`NameHome` is a sub type of `Name`)

`NameMobile <: Name` (`NameMobile` is a sub type of `Name`)

```
def greeting(r: Name) = "Hi " + r.name + ", How are you?"
greeting(tom)
greeting(bob)
```

# Sub Types

- The sub type relation is kind of the subset relation.
- But they are NOT the same.

- T <: S
  Every element of T can be used as that of S.

- *Cf*. T is a subset of S.
  Every element of T is that of S.

- Why polymorphism?
  A function of type S=>R can be used as T=>R for many sub types T of S.
  Note that S=>R <: T=>R when T <: S.

# Two Kinds of Sub Types

- Structural Sub Types
  - The system implicitly determines the sub type relation by the structures of data types.
  - Structurally equivalent types are the same.

- Nominal Sub Types
  - The user explicitly specify the sub type relation using the names of data types.
  - Structurally equivalent types with different names may be different.

# Structural Sub Types

# General Sub Type Rules

- Reflexivity:
  For any type T, we have:

$$T <: T$$

- Transitivity:
  For any types T, S, R, we have:

$$T <: R \qquad\qquad R <: S$$

=======================

$$T <: S$$

# Sub Types for Special Types

- Nothing: The empty set
- Any: The set of all values

- For any type T, we have:

  Nothing <: T <: Any

- Example

```
val a : Int = 3
val b : Any = a
def f(a: Nothing) : Int = a
```

# Sub Types for Records

- Permutation

$$\frac{}{\{\ldots;\ x: T1;\ y: T2;\ \ldots\} <: \{\ldots;\ \ y: T2,\ x: T1;\ \ldots\ \}}$$

- Width

$$\frac{}{\{\ldots;\ x: T;\ \ldots\} <: \{\ \ldots;\ \ldots\ \}}$$

- Depth

$$\frac{T <: S}{\{\ldots;\ \ x: T\ ;\ \ldots\} <: \{\ \ldots;\ x: S;\ \ldots\ \}}$$

# Sub Types for Records

- Example

  {val x: { val y: Int; val z: String}, val w: Int}

  <: (by permutation)

  {val w: Int; val x: { val y: Int; val z: String}}

  <: (by depth & width)

  {val w: Int; val x: {val z: String}}

# Sub Types for Functions

- Function Sub Type

$$\frac{T <: T' \qquad S <: S'}{(T' \Rightarrow S) <: (T \Rightarrow S')}$$

- Example

```
def foo(s: {val a: Int; val b: Int}) :
  {val x: Int; val y: Int} = {
  object tmp {
    val x = s.b
    val y = s.a
  }
  tmp
}
val gee:
  {val a: Int; val b: Int; val c: Int} =>
  {val x: Int} =
  foo _
```