Control Abstraction

Dr.. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

09, 2013

Outline

1. **Subprogram Definition**
2. **Subprogram Mechanisms**
   - Simple Call Return
   - Recursie Call
   - Exception
3. **Parameter Passing**
4. **Higher-order Funtions**

Subprogram Definition

Subprogram definition consists of:

- **Specification**
  - Subprogram name
  - Parameters
    - input + output
    - order
    - type
    - parameter passing mechanisms: by value, by reference, by name,...
  - Behaviour of the subprogram
- **Implementation**:
  - Local data
  - Collection of statements as **subprogram body**

Example

```
def apply(interval: Int,
          repeats: Boolean = true)
          (op: => Unit) {
  val timeOut = new javax.swing.AbstractAction() {
    def actionPerformed
    (e: java.awt.event.ActionEvent) = op
  }
  val t = new javax.swing.Timer(interval, timeOut)
  t.setRepeats(repeats)
  t.start()
}
```

1. How many subprogram definitions are there in the above code?
2. How many parameters are there in each subprogram definition?

Subprogram Activation

An activation of a subprogram:

- is created when the subprogram is invoked
- is destroyed when the subprogram completed its execution

An activation includes

- Static part: Code segment
- Dynamic part: Activation record
  - formal parameters
  - local data
  - return address
  - other links

Subprogram Mechanisms

- Simple Call-Return
- Recursive Call
- Exception Processing Handler
- Coroutines
- Scheduled Subprograms
- Tasks

Simple Call-Return

Basic Features

- No recursion
- Explicit Call Site
- Single Entry Point
- Immediate Control Passing
- Single Execution

Recursive Call

- Be able to call recursive
  - Direct Recursice Call
  - Indirect Recursive Call (Mutual Recursive)
- Other features same as Simple Call-Return

Exception Processing Handler

- May have no explicit call site
- Used in
  - Event-Driven Programming
  - Error Handler

Example,

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp(){
    if (length(V)==0) throw new EmptyExcp();
    else ...
};
...
try{...
   average(W);
   ...
}
catch (EmptyExcp e) {write("Array empty"); }
```

Exception Mechanisms

A language must specify:

- *which* exceptions can be handled and *how* they can be defined
- *how* an exception can be raised
- *how* an exception can be handled

How an exception can be defined

- Java: subclass of *Throwable*
- Ada: values of a special type
- C++: any value

How an exception can be raised

Raising exception

- By user interaction
  (Click, MouseMove, TextChange, ...)
- By operating system
- By an object (Timer)
- By programmer (throw)

Example in Scala

```
object Timer {
  def apply(interval: Int,
            repeats: Boolean = true)
            (op: => Unit) {
    val timeOut = new javax.swing.AbstractAction() {
      def actionPerformed
      (e: java.awt.event.ActionEvent) = op
    }
    val t = new javax.swing.Timer(interval, timeOut)
    t.setRepeats(repeats)
    t.start()
  }
}
Timer(2000) { println("Timer went off") }
Timer(10000, false) { println("10 seconds are over!") }
```

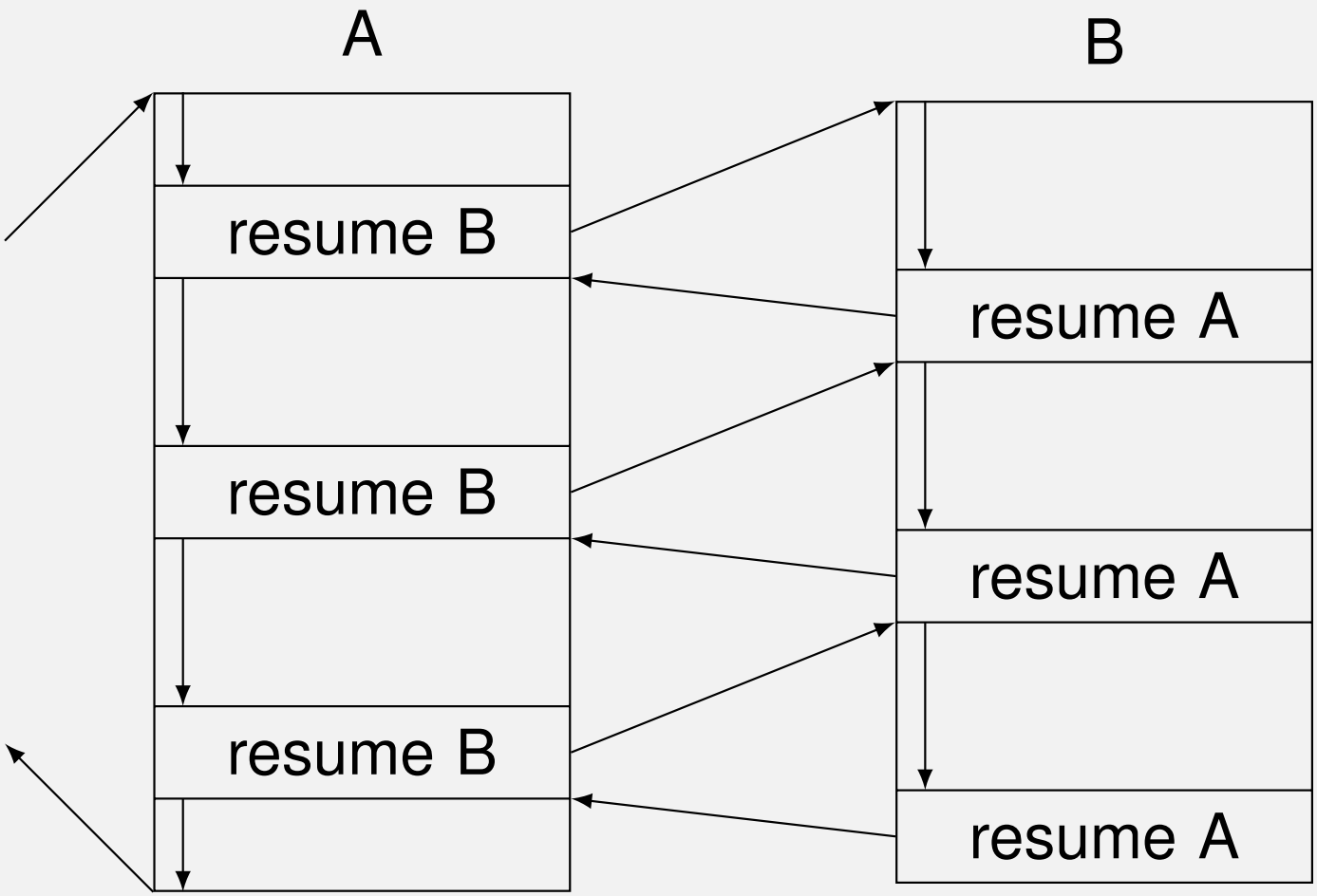
How an exception can be handled

- Define *the protected block* to intercept the exception for being handled
- Define exception handler associated with the protected block

Termination Semantic

- non-resumable (common) + stack unwinding
- resumable
  - at the statement causing the error
  - after the statement causing the error

A coroutine may postpone its execution and control is back to caller. Its execution later is resumed at the place it postponed.



Tasks

- able to execute concurrently with other tasks
- run on multi-processor machine or
- single processor machine using time sharing

Issue?

- Synchronization
  - Race condition
  - Deadlock
- Communication

Example in Scala

```
val pa = (0 until 10000).toArray.par

pa.map(_ + 1)

pa map { v => if (v % 2 == 0) v else -v }

pa.fold(0) { _ + _ }

var a = 0

pa foreach { a += _ }
```

Scheduled subprograms

- The execution of callee is NOT started when it is invoked
  - scheduled by time
    CALL A AT TIME = CURRENT_TIME + 10
  - scheduled by priority
    CALL B WITH PRIORITY 7
- Controlled by a scheduler

Formal and Actual Parameter

Definition

- Formal parameters:int foo(float x,bool& y);
  - just a simple name
  - close to a variable declaration
  - combine with symbols relating to parameter passing mechanism
- Actual parameters/Arguments:foo(4*a,b)
  - an expression

Formal-Actual Corresponding

- by position
  int foo(float a,int b) $\Leftarrow$ foo(x+1,y-2)
- by name
  int foo(float a,int b) $\Leftarrow$ foo(b = x+1, a = y-2)

Parameter Passing

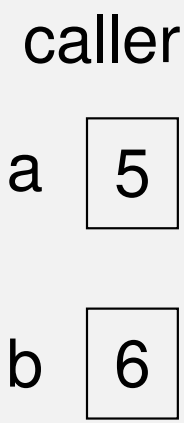
- Input-Output
  - By value-result
  - By reference
  - By name
- Input Only
  - By value
  - By constant reference
- Output Only
  - By result
  - As a result of a function

Input-Output Parameters

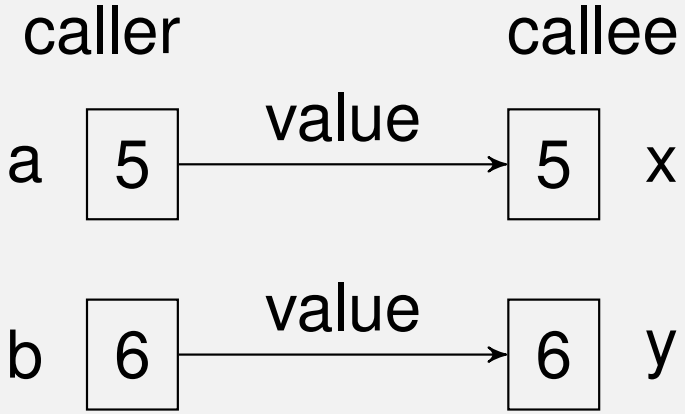
- Pass by value-result

Input-Output Parameters

- Pass by value-result



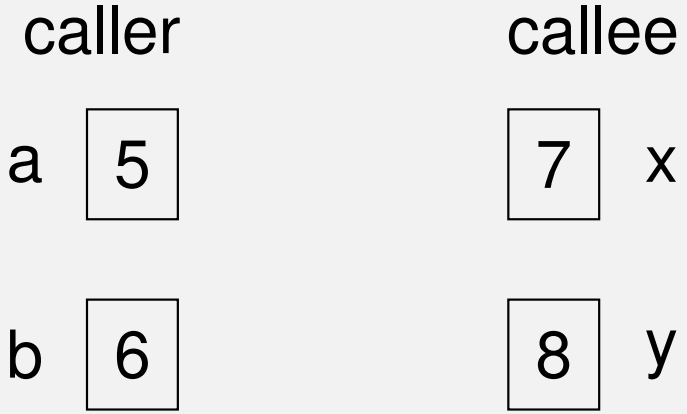
- Pass by value-result
  findMax(a,b) $\Rightarrow$ **int** findMax(**int** x,**int** y) {...}



Input-Output Parameters

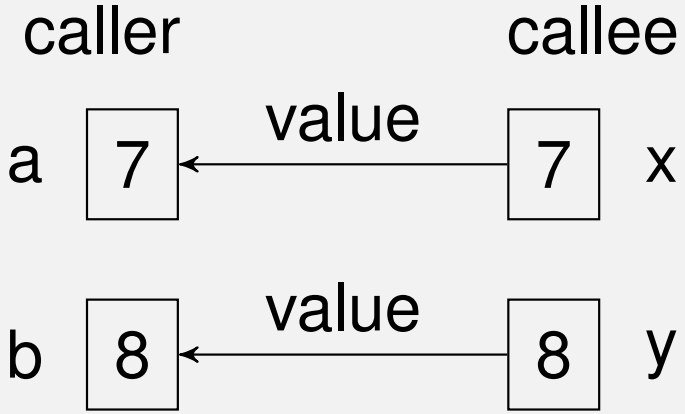
- Pass by value-result

int findMax(**int** x,**int** y) {...}



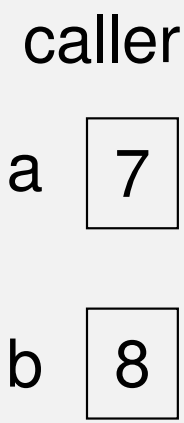
Input-Output Parameters

- Pass by value-result
  findMax(a,b) $\Leftarrow$ **int** findMax(**int** x,**int** y) {...}



Input-Output Parameters

- Pass by value-result



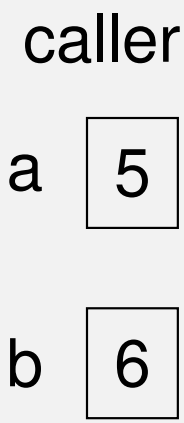
Input-Output Parameters

- Pass by value-result
- Pass by reference

Input-Output Parameters

- Pass by value-result

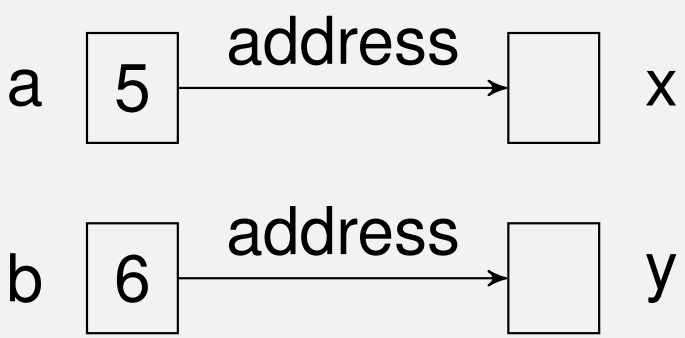
- Pass by reference



Input-Output Parameters

- Pass by value-result

- Pass by reference
  findMax(a,b) $\Rightarrow$ **int** findMax(**int**& x,**int**& y) {...}

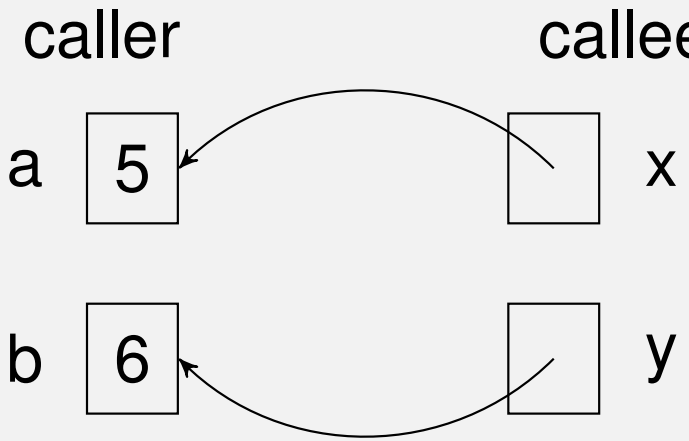


Input-Output Parameters

- Pass by value-result

- Pass by reference

int findMax(**int**& x,**int**& y) {...}

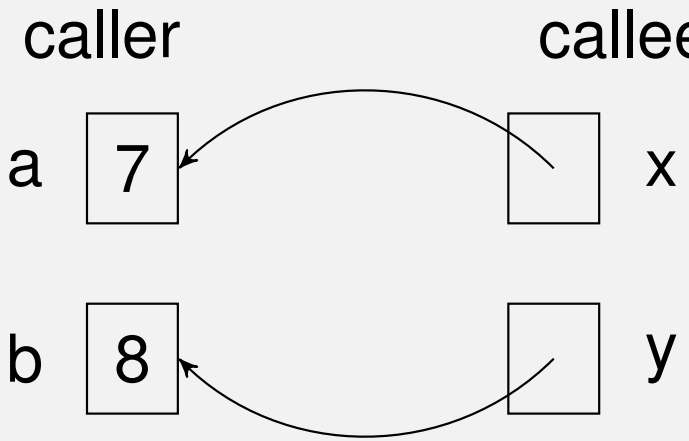


Input-Output Parameters

- Pass by value-result

- Pass by reference

int findMax(**int**& x,**int**& y) {...}



Input-Output Parameters

- Pass by value-result

- Pass by reference
  findMax(a,b) $\Leftarrow$
  caller

  a | 7 |

  b | 8 |

Input-Output Parameters

- Pass by value-result
- Pass by reference
- Pass by name

Input-Output Parameters

- Pass by value-result
- Pass by reference
- Pass by name
  findMax(a,b) $\Rightarrow$ **int** findMax(**int**$\Rightarrow$ x,**int**$\Rightarrow$ y) {...}

Input-Output Parameters

- Pass by value-result
- Pass by reference
- Pass by name

int findMax(**int**⇒ x,**int**⇒ y) {...}

a ≡ x

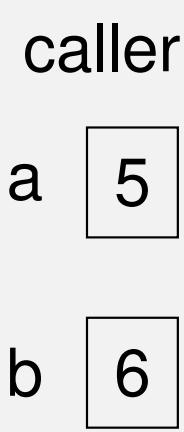
b ≡ y

Input Only Parameters

- Pass by value

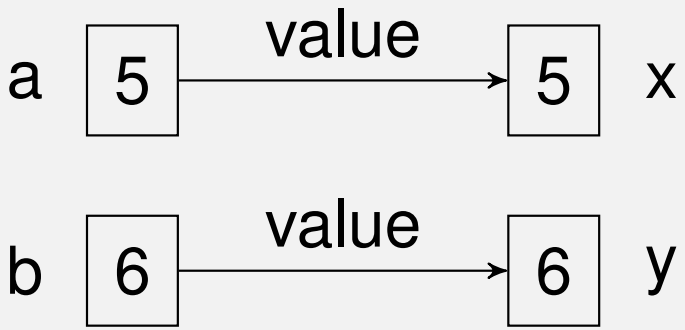
Input Only Parameters

- Pass by value



Input Only Parameters

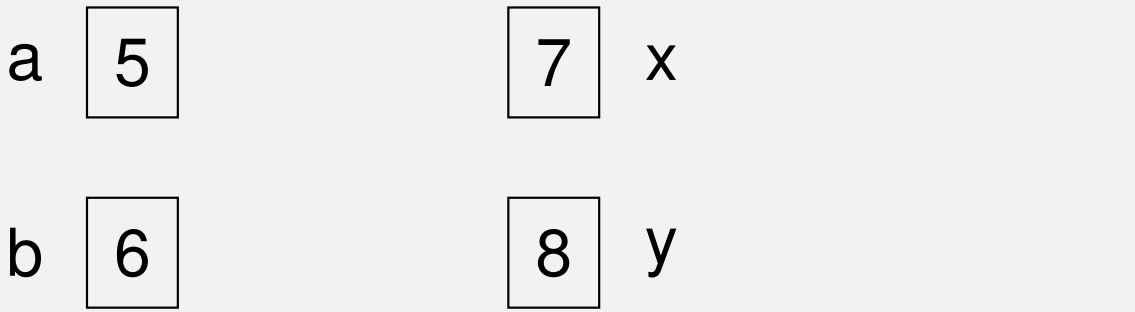
- Pass by value
  findMax(a,b) $\Rightarrow$ **int** findMax(**int** x,**int** y) {...}



Input Only Parameters

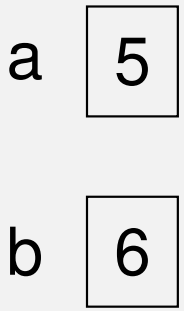
- Pass by value

int findMax(**int** x,**int** y) {...}



Input Only Parameters

- Pass by value
  findMax(a,b) $\Leftarrow$
  caller



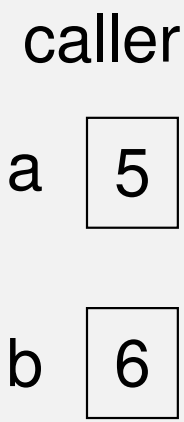
Input Only Parameters

- Pass by value
- Pass by constant reference

Input Only Parameters

- Pass by value

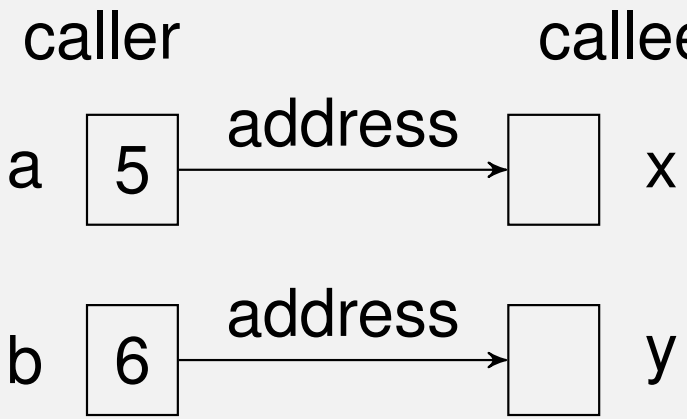
- Pass by constant reference



Input Only Parameters

- Pass by value

- Pass by constant reference
  findMax(a,b) $\Rightarrow$ **int** findMax(**const int**& x,**const int**& y) {...}



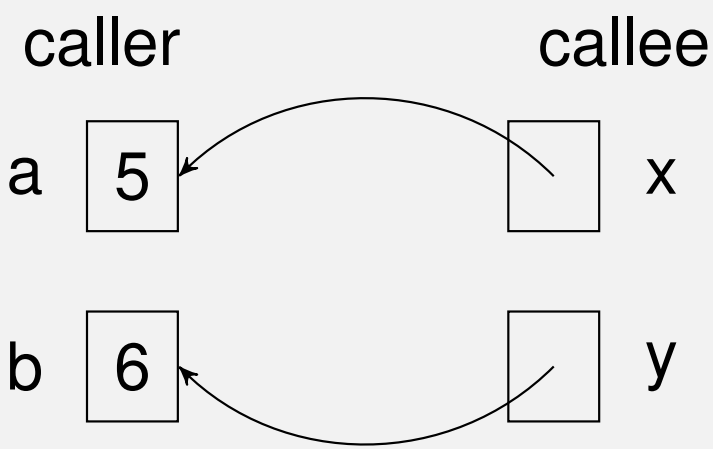
Input Only Parameters

- Pass by value

- Pass by constant reference

int findMax(**const int**& x,**const int**& y)

{...}



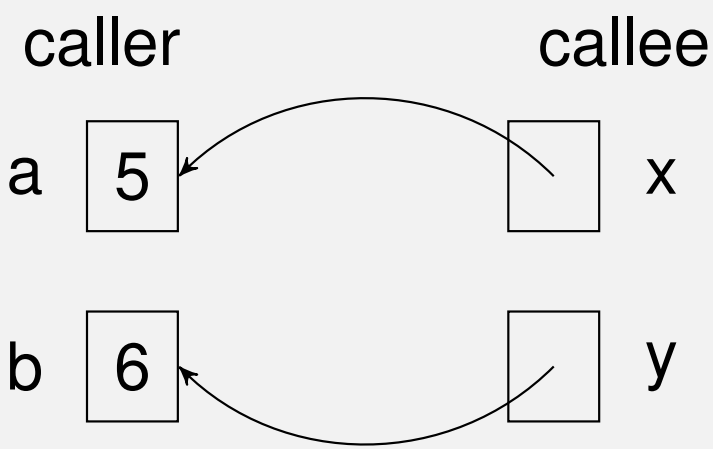
Input Only Parameters

- Pass by value

- Pass by constant reference

int findMax(**const int**& x,**const int**& y)

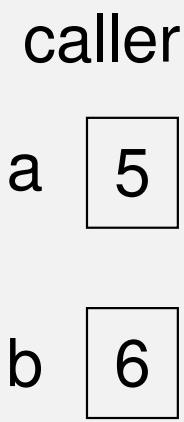
{...}



Input Only Parameters

- Pass by value

- Pass by constant reference
  findMax(a,b) $\Leftarrow$



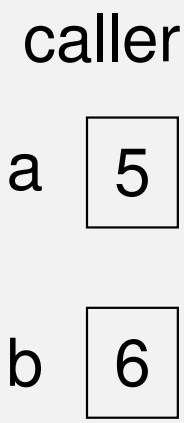
Output Only Parameters

- Pass by result

- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Output Only Parameters

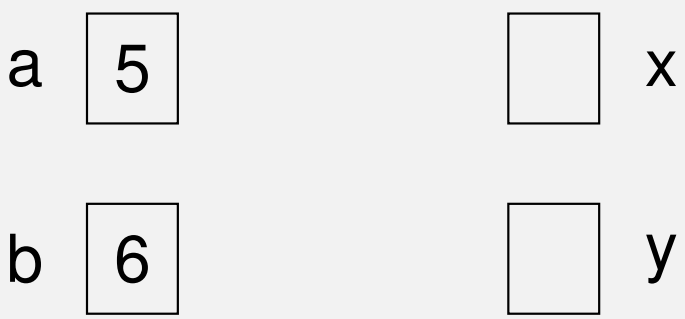
- Pass by result



- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Output Only Parameters

- Pass by result
  findMax(a,b) $\Rightarrow$ **int** findMax(**int** x,**int** y) {...}

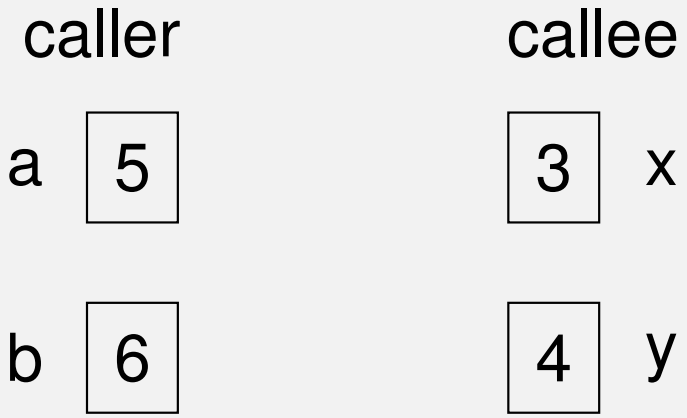


- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Output Only Parameters

- Pass by result

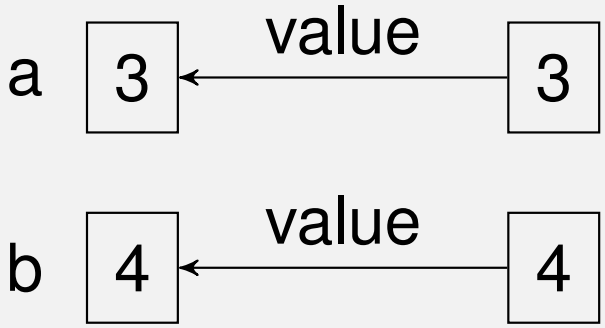
int findMax(**int** x,**int** y) {...}



- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Output Only Parameters

- Pass by result
  findMax(a,b) $\Leftarrow$
  caller



- As a result of a function
  int foo() ... return 0;
  No actual parameter: foo() + 1

Higher-order Functions

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

Example, in stdlib.h of C, there is a built-in sorting function

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));

int
int_sorter(const void *first_arg ,const void *second_arg)
    int first = *(int*)first_arg;
    int second = *(int*)second_arg;
    if (first < second)  return -1;
    else if (first == second) return 0;
    else return 1;
}
```

Higher-order Functions

A function is *higher-order* when it accepts functions

- as its input parameters (fairly common)
- as its out parameters (less common - but required in functional programming)

Example, in stdlib.h of C, there is a built-in sorting function

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));

int main() {
    int array[10],i;
    /* fill array */
    for ( i = 0; i < 10; ++i )
        array[i] = 10 - i;
    qsort(array, 10, sizeof(int),int_sorter);
    for (i = 0; i < 10; ++i)
        printf ("%d\n" ,array[i]);
}
```

What is non-local environment?

- Deep binding
- Shallow binding

Example, Static scope: z = 6

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}
...
{int x = 4;
 int z = g(f); //deep binding
}
```

▸ Skip Scala Example

What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Deep binding: z = 9

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}
...
{int x = 4;
 int z = g(f); //deep binding
}
```

▸ Skip Scala Example

What is non-local environment?

- Deep binding
- Shallow binding

Example, Dynamic scope + Shallow binding: z = 7

```
int x = 1;
int f(int y){ return x+y; }

int g (int h(int b)){
    int x = 2;
    return h(3) + x; //shallow binding
}
...
{int x = 4;
 int z = g(f); //deep binding
}
```

▸ Skip Scala Example

Example in Scala

```
object FileMatcher {
    private def filesHere =
          (new java.io.File(".")).listFiles

    def filesEnding(query: String) =
      for (file <- filesHere;
           if file.getName.endsWith(query))
        yield file}

    def filesContaining(query: String) =
        for (file <- filesHere;
             if file.getName.contains(query))
          yield file

    def filesRegex(query: String) =
       for (file <- filesHere;
            if file.getName.matches(query))
          yield file
  }
```

Example in Scala

```
object FileMatcher {
    private def filesHere =
        (new java.io.File(".")).listFiles

    def filesMatching(query: String,
        matcher: (String, String) => Boolean) = {
      for (file <- filesHere;
          if matcher(file.getName, query))
        yield file
    }
    def filesEnding(query: String) =
        filesMatching(query, _.endsWith(_))

    def filesContaining(query: String) =
        filesMatching(query, _.contains(_))

    def filesRegex(query: String) =
        filesMatching(query, _.matches(_))
}
```

What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

What returns as functions

- Code
- Environment

Example,

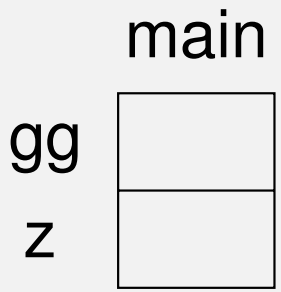
```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

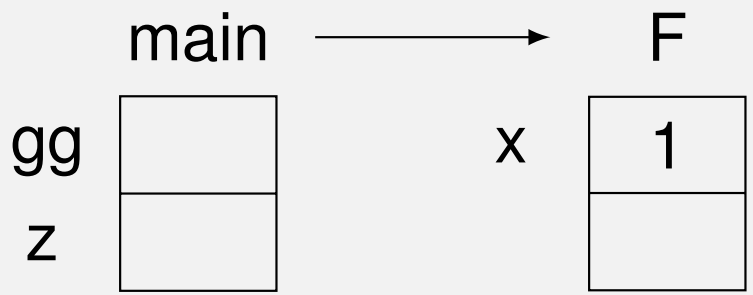


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

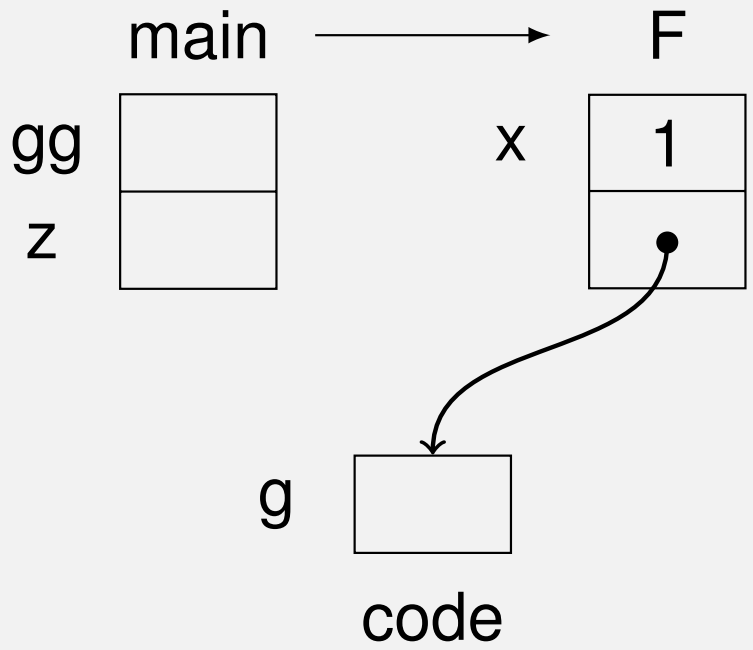


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

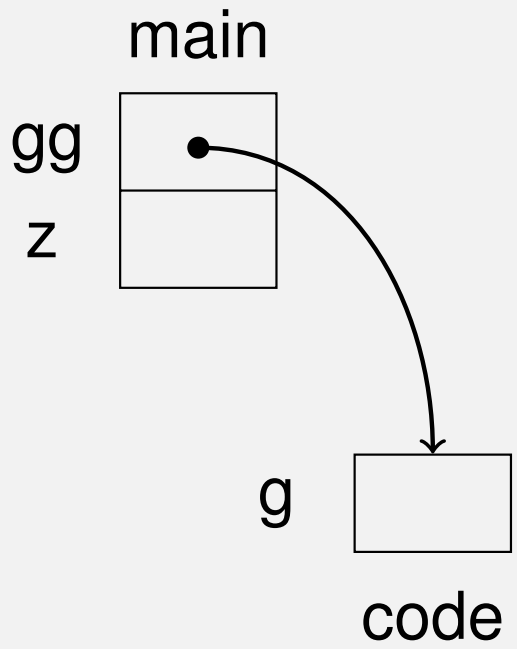


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

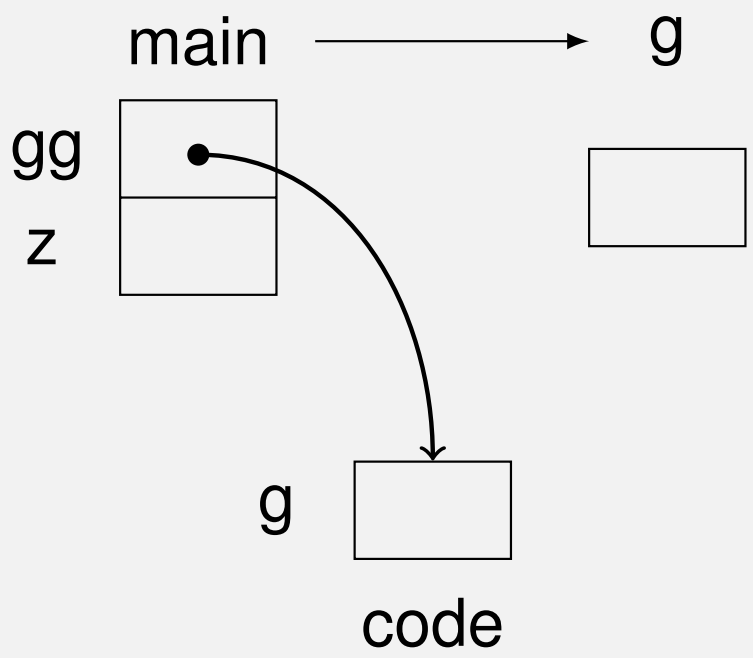


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```

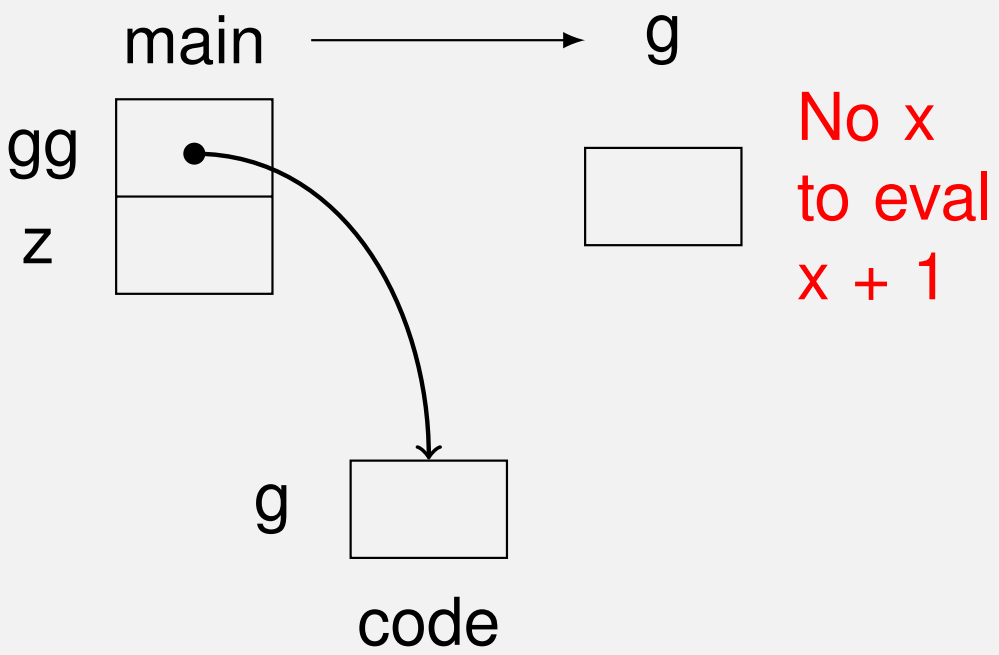


What returns as functions

- Code
- Environment

Example,

```
void->int F () {
    int x = 1;
    int g () {
        return x+1;
    }
    return g;
}
void->int gg = F();
int z = gg();
```



Summary

- Subprogram mechanisms
  - Simple Call-Return
  - Recursive Call
  - Exception
  - Coroutine
  - Scheduled Call
  - Tasks
- Parameter Passing
- Higher-order Functions