Topics

- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Design Issues for Functions
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded Operators
- Closures
- Coroutines

Parameters that are Subprogram Names

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

Parameters that are Subprogram Names: Referencing Environment

- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- *Deep binding*: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

Referencing Environment: Example

```
function sub1() {
  var x;
  function sub2() {
    alert(x);  // Creates a dialog box with the value of x
    };
  function  sub3() {
    var  x;
    x = 3;
    sub4(sub2);
    };
  function  sub4(subx) {
    var  x;
    x = 4;
    subx();
    };
  x = 1;
  sub3();
  };
```

Calling Subprograms Indirectly

- Usually when there are several possible subprograms to be called and the correct one on a particular run of the program is not know until execution (e.g., event handling and GUIs)
- In C and C++, such calls are made through function pointers

Calling Subprograms Indirectly (continued)

- In C#, method pointers are implemented as objects called *delegates*
  - A delegate declaration:

```
public delegate int Change(int x);
```

    - This delegate type, named `Change`, can be instantiated with any method that takes an **`int`** parameter and returns an **`int`** value

    A method: `static int fun1(int x) { … }`

    Instantiate: `Change chgfun1 = new Change(fun1);`

    Can be called with: `chgfun1(12);`

  - A delegate can store more than one address, which is called a *multicast delegate*

Design Issues for Functions

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class

Overloaded Subprograms

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms, e.g. constructors
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

Generic Subprograms

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*
  - A cheap compile-time substitute for dynamic binding

Generic Subprograms (continued)

- C++
  - Generic subprograms are preceded by a **`template`** clause that lists the generic variables, which can be type names or class names

```
template <class Type>
  Type max(Type first, Type second) {
  return first > second ? first :
second;
  }
```

Generic Subprograms (continued)

- Java 5.0
  - Differences between generics in Java 5.0 and those of C++:
  1. Generic parameters in Java 5.0 must be classes
  2. Java 5.0 generic methods are instantiated just once as truly generic methods
  3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
  4. Wildcard types of generic parameters

Generic Subprograms (continued)

- Java 5.0 (continued)

```
public static <T> T doIt(T[] list) { … }
```

  - The parameter is an array of generic elements (`T` is the name of the type)
  - A call:

```
doIt<String>(myList);
```

Generic parameters can have bounds:

```
public static <T extends Comparable> T
  doIt(T[] list) { … }
```

The generic type must be of a class that implements the `Comparable` interface

Generic Subprograms (continued)

- Java 5.0 (continued)
  - Wildcard types

    `Collection<?>` is a wildcard type for collection classes

```
void printCollection(Collection<?> c) {
   for (Object e: c) {
     System.out.println(e);
   }
 }
```

  - Works for any collection class

Generic Subprograms (continued)

- C# 2005
  - Supports generic methods that are similar to those of Java 5.0
  - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
    - Another – C# 2005 does not support wildcards

Generic Subprograms (continued)

- F#
  - Infers a generic type if it cannot determine the type of a parameter or the return type of a function – *automatic generalization*
  - Such types are denoted with an apostrophe and a single letter, e.g., `'a`
  - Functions can be defined to have generic parameters

```
let printPair (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

    - `%A` is a format code for any type
    - These parameters are not type constrained

Generic Subprograms (continued)

- F# (continued)
  - If the parameters of a function are used with arithmetic operators, they are type constrained, even if the parameters are specified to be generic
  - Because of type inferencing and the lack of type coercions, F# generic functions are far less useful than those of C++, Java 5.0+, and C# 2005+

User-Defined Overloaded Operators

- Operators can be overloaded in Ada, C++, Python, and Ruby
- A Python example

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

Use: To compute `x + y`, `x.__add__(y)`

Closures

- A *closure* is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - A static-scoped language that does not permit nested subprograms doesn't need closures
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
  - To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

A static-scoped language that does not permit nested subprograms doesn't need closures

```
#include <stdio.h>
int x = 10;
int f() {
        return x;
}
int g() {
        int x = 20;
        return f();
}
int main() {
        printf("%d\n", g());
        return 0;
}
```

A JavaScript Closure

```
function init() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  displayName();
}
init();
```

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

Closures (continued)

- A JavaScript closure:

```
function makeAdder(x) {
  return function(y) {return x + y;}
}
...
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) +
               "<br />");
document.write("add 5 to 20: " + add5(20) +
               "<br />");
```

  - The closure is the anonymous function returned by `makeAdder`

Closures (continued)

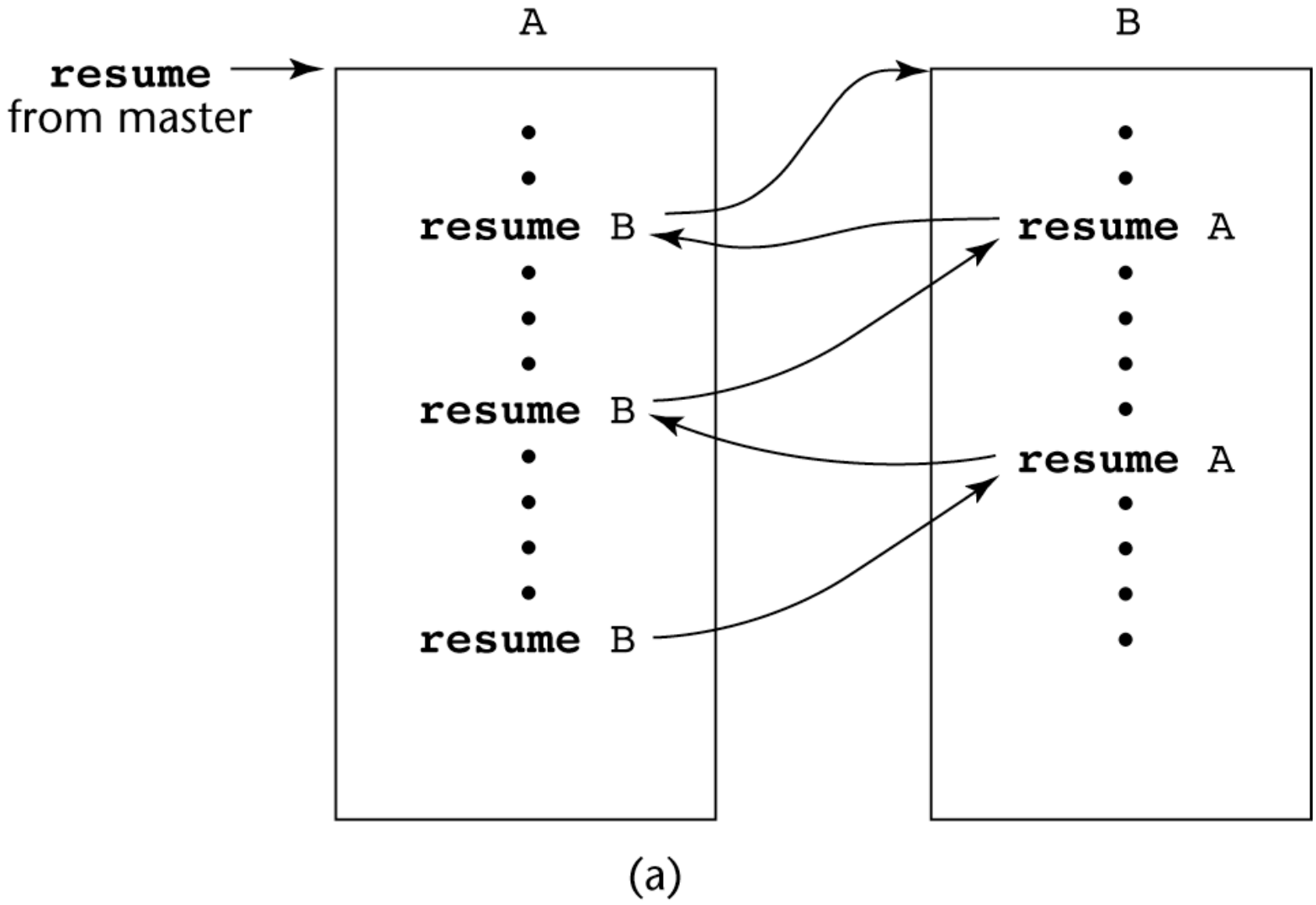
- C#
  - We can write the same closure in C# using a nested anonymous delegate
  - `Func<int, int>` (the return type) specifies a delegate that takes an `int` as a parameter and returns and `int`

```
static Func<int, int> makeAdder(int x) {
   return delegate(int y) {return x + y;};
}
...
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

Coroutines

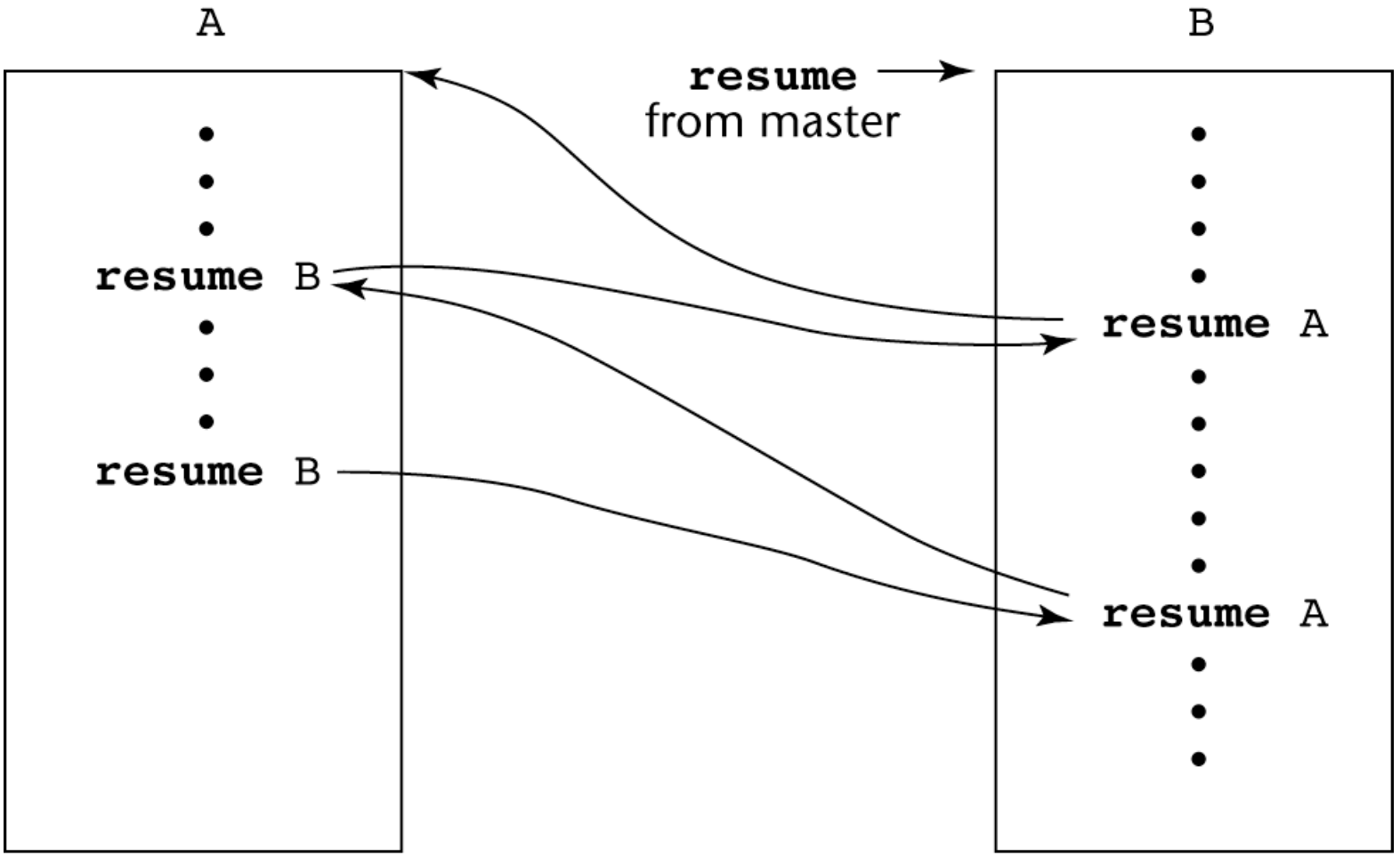
- A *coroutine* is a subprogram that has multiple entries, depending on when it is called – supported directly in Lua
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

Coroutines Illustrated: Possible Execution Controls



(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops