Organization of Programming Languages CS 3200/5200N

Lecture 09

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Control Flow

- **Control flow** = the flow of control, or execution sequence, in a program.

- Levels of control flow:
  1. Within **expressions**.
  2. Among **program statements**.
  3. Among **program units**.

Abstraction

- Abstraction = a view or representation of an entity that includes only the most significant attributes.

- Birds are animals with the following attributes:
  - two wings
  - two legs
  - a tail, feathers, …
- Robins are birds that …
- Sparrows are birds that …

⇒ significant simplification of descriptions of members.

Abstraction

- Two fundamental abstraction facilities in PLs:
  - **Process abstraction**:
    - Emphasized from early days.
    - Abstract away the details of the implementation by using just a call statement.
  - **Data abstraction**:
    - Emphasized in the 1980s.
    - Abstract away from the type representation and the implementation details of its operations by using an *abstract data type*.

Subprograms: General Characteristics

- Each subprogram has a single entry point:
  - Exception: coroutines.
- The calling program is suspended during execution of the called subprogram:
  - Exception: concurrent units.
- Control always returns to the caller when the called subprogram's execution terminates.

Subprograms: Procedures vs. Functions

- A **procedure** is a named scope that is parameterized:
  - **Procedure body**: defines a scope that contains local variable type declarations and statements.
  - **Parameters**: allow additional values, variable references, or names to be bound into the scope, depending on the *calling convention semantics*.
    - formal parameters when the procedure is defined.
    - actual parameters when the procedure is called.
  - **Name**: may be overloaded to have different meaning depending on the type of the arguments

Subprograms: Procedures vs. Functions

- A **function** structurally resembles a procedure, but is semantically modeled on mathematical functions:
    - Functions are expected to produce no side effects.
    - Functions are required to produce a return value.
    - Functions should have at least one argument.

- In some languages (e.g. C/C++) the terms **function** and **procedure** are used interchangeably:
    - a distinction should be made.
    - examples of procedure vs. function behavior in C/C++.

Basic Definitions

- A **subprogram definition** describes the *interface* to and the *actions* of the subprogram abstraction:
  - Ada and Fortran also specify the type of the subprogram:

    ```
    procedure Adder(parameters)
    ```

  - Other languages have only one kind of subprogram (functions).

- In Python, subprogram definitions are executable statements:

```
if … :
    def fun(…):
        …
else :
    def fun(…):
        …
```

Basic Definitions

- A **subprogram header** is the first part of the definition, including:
  - the kind of subprogram;
  - the name (can be overloaded);
  - the formal parameters.

- The **parameter profile** (i.e. **signature**) of a subprogram is the number, order, and types of its parameters.

- The **protocol** of a subprogram is:
  - a parameter profile for procedures.
  - a parameter profile + its return type for functions.

Basic Definitions

- A **subprogram declaration** provides the protocol, but not the body, of the subprogram.

- A **subprogram call** is an explicit request to execute the subprogram:
  - **actual parameters** are mapped to corresponding **formal parameters** based on *corespondence rules* of the language.
  - **actual parameters** are bound to **formal parameters** based on the *calling convention semantics*.

Actual/Formal Parameter Correspondence

- **Positional**:
  - The first actual parameter is bound to the first formal parameter, and so forth.
  - Safe and effective.
  - Nearly all programming languages.
- **Keyword**:
  - The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.
  - *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors.
  - *Disadvantage*: User must know the formal parameter's names
  - Ada, Fortran95, Python.

Formal Parameters: Default Values

- In certain languages formal parameters can have default values (if no actual parameter is passed):
  - Examples: C++, Python, Ruby, Ada, PHP.

- In C++, default parameters must appear last because parameters are positionally associated.

- In Python, *it used to be the case that* default parameters can appear at any position:
  - all actual parameters after the absent one must be keyworded.

```
def compute_pay(income, exemptions = 1, tax_rate):
...
pay = compute_pay(20000, tax_rate = 0.15)
```

Local Referencing Environment

- **Local referencing environment** is defined by:
  - local variables;
  - formal parameters;
- Local variables can be stack-dynamic or static:
  - Advantages of stack-dynamic:
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages of stack-dynamic:
    - Allocation/de-allocation, initialization time
    - Indirect addressing
    - Subprograms cannot be history sensitive

Local Referencing Environment

- C/C++ example:

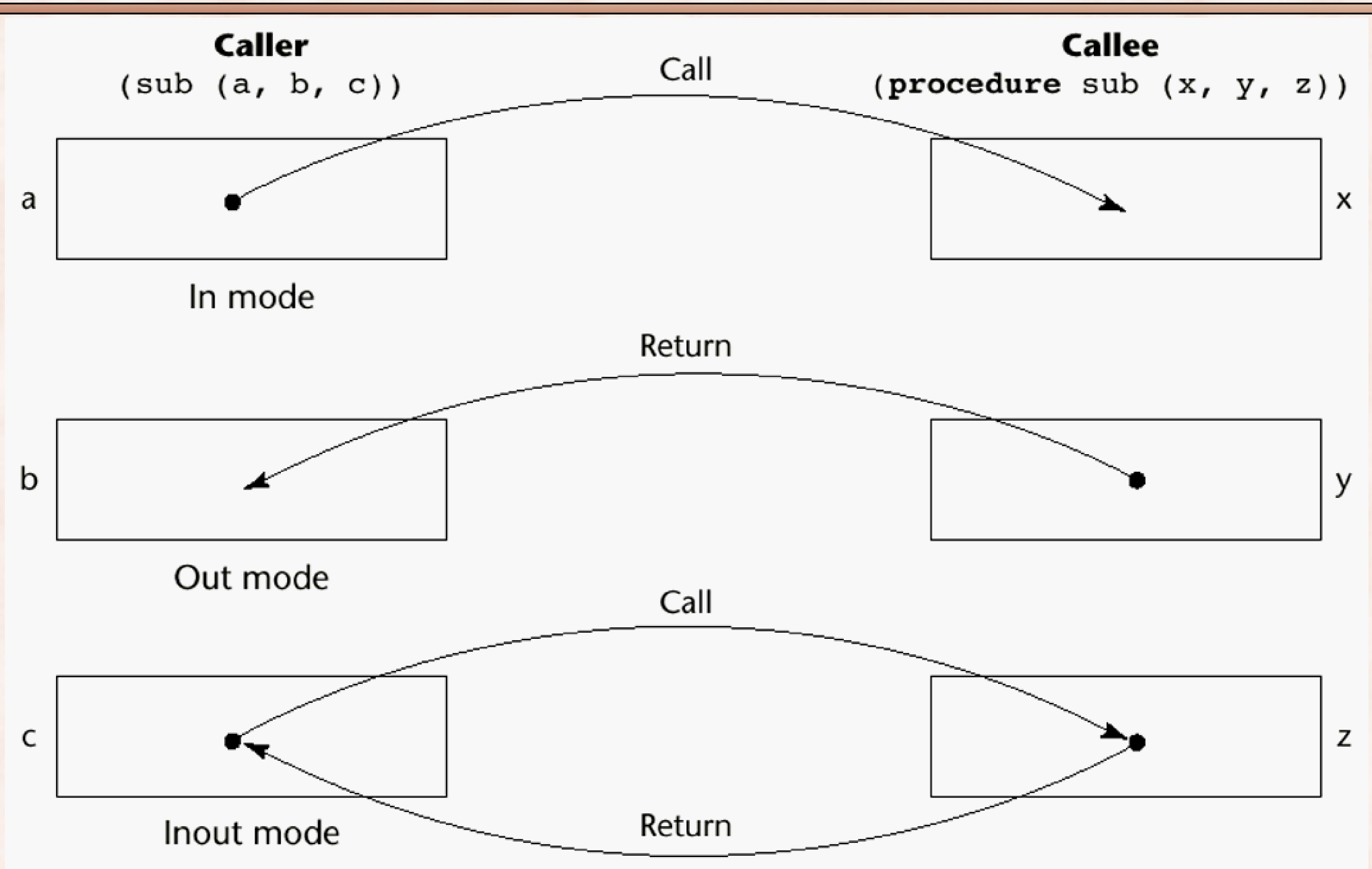
```
int length (const char* string) {
    int len = 0;
    if (string == NULL || *string == '\0')
        return 0;
    while (*string != '\0') {
        len++;
        string++;
    }
    return len;
}
```

- When the function is called, its environment is activated:
  - bindings of local variables to stack locations (l-values).
  - bindings of argument values (r-values) to stack locations associated with formal parameters.

Semantic Models of Parameter Passing

- **In mode**: *formal parameters* can receive data from the corresponding *actual parameters*.
- **Out mode**: *formal parameters* can transmit data to the corresponding *actual parameters*.
- **InOut mode**: both.

Semantic Models of Parameter Passing: x **in**, y **out**, z **in out**



Models of Parameter Passing: Implementations (Calling Conventions)

- **Pass-by-Value** (`In` Mode)
- **Pass-by-Result** (`Out` Mode)
- **Pass-by-Value-Result** (`InOut` Mode)
- **Pass-by-Reference** (`InOut` Mode, but also `In`, or `Out`)
- **Pass-by-Name** (`InOut` Mode)

- In most languages parameter communication takes place through the run-time stack.
  - Copy an actual value (r-value)
  - Transfer an access path (l-value)
    - a pointer, or a reference.

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter:
  - The r-value of actual param is copied on the stack (pass by copy).
    - Disadvantages:
      - additional storage is required (stored twice) and the actual move can be costly (for large parameters).

Pass-by-Value (In Mode): Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   function Fun_In(X: in Integer) return Integer is
   begin
      X := X + 3;
      return 2 * X;
   end Fun_In;

   A: Integer := 10;

begin
   Put(Fun_In(A));
   Put(A);
end Parameter_Test;
```

```
> gnat make parameter_test.adb
gcc-4.4 -c parameter_test.adb
parameter_test.adb:13:07: assignment to "in" mode
                          parameter not allowed
gnatmake: "parameter_test.adb" compilation error
```

Pass-by-Value (In Mode): Ada

```ada
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   function Fun_In(X: in Integer) return Integer is
   begin
      return 2 * X;
   end Fun_In;

   A: Integer := 10;

begin
   Put(Fun_In(A));
   Put(A);
end Parameter_Test;
```

```
> gnat make parameter_test.adb
gcc-4.4 -c parameter_test.adb
gnatbind -x parameter_test.ali
gnatlink parameter_test.ali

> ./parameter_test
        20          10
```

Pass-by-Result (Out Mode)

- No value is transmitted to the subprogram.
- The formal parameter acts as a local variable.
- Before control is returned to the caller, the value of the formal parameter is transmitted back to the caller's actual parameter:
  - Actual parameter must be a variable.
  - Implemented by copying an r-value from the stack.
- Potential problem:
  - `sub(p1, p1);`
    - whichever formal parameter is copied back will represent the current value of `p1`.

Pass-by-Result (Out Mode): Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   procedure Fun_Out(X: out Integer) is
   begin
      X := 2 * X;
   end Fun_Out;

   A: Integer := 10;

begin
   Fun_Out(A);
   Put(A);
end Parameter_Test;
```

```
> gnat make parameter_test.adb
gcc-4.4 -c parameter_test.adb
parameter_test.adb:8:16: warning: "X" may be
                    referenced before it has a value
gnatbind -x parameter_test.ali
gnatlink parameter_test.ali

> ./parameter_test
      0
```

Pass-by-Result (Out Mode): Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   procedure Fun_Out(X: out Integer) is
   begin
      X := 3;
      X := 2 * X;
   end Fun_Out;

   A: Integer := 10;

begin
   Fun_Out(A);
   Put(A);
end Parameter_Test;
```

```
> gnat make parameter_test.adb
gcc-4.4 -c parameter_test.adb
gnatbind -x parameter_test.ali
gnatlink parameter_test.ali

> ./parameter_test
        6
```

Pass-by-Result (Out Mode): Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   procedure Fun_Out(X: out Integer) is
   begin
      X := 3;
      X := 2 * X;
   end Fun_Out;

   A: Integer := 10;

begin
   Fun_Out(5);
   Put(A);
end Parameter_Test;
```

[razvan@texas ada]$ *gnat make parameter_test.adb*
gcc-4.4 -c parameter_test.adb
parameter_test.adb:14:08: actual for "X" must be a variable
gnatmake: "parameter_test.adb" compilation error

Pass-by-Value-Result (InOut Mode)

- A combination of *pass-by-value* and *pass-by-result*.
- Sometimes called *copy-in/copy-out*.
- Formal parameters have local storage.
- Semantically similar to *pass-by-reference*:
  - There may be subtle differences due to aliasing in pass-by-reference!

Pass-by-Value-Result (InOut Mode): Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   procedure Fun_InOut(X: in out Integer) is
   begin
      X := 2 * X + 1;
   end Fun_InOut;

   A: Integer := 10;

begin
   Fun_InOut(A);
   Put(A);
end Parameter_Test;
```

```
> gnat make parameter_test.adb
gcc-4.4 -c parameter_test.adb
gnatbind -x parameter_test.ali
gnatlink parameter_test.ali

> ./parameter_test
        21
```

Functions vs. Procedures: Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   procedure Fun_InOut(X: in out Integer) return Integer is
   begin
      X := 2 * X;
      return X - 1;
   end Fun_InOut;

   A: Integer := 10;

begin
   Fun_InOut(A);
   Put(A);
end Parameter_Test;
```

> *gnat make parameter_test.adb*
gcc-4.4 -c parameter_test.adb
parameter_test.adb:5:04: "procedure" should be "function"
gnatmake: "parameter_test.adb" compilation error

Functions vs. Procedures: Ada

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Parameter_Test is

   function Fun_InOut(X: in out Integer) return Integer is
   begin
      X := 2 * X;
      return X - 1;
   end Fun_InOut;

   A: Integer := 10;

begin
   Fun_InOut(A);
   Put(A);
end Parameter_Test;
```

```
> gnat make parameter_test.adb
gcc-4.4 -c parameter_test.adb
parameter_test.adb:5:21: functions can only have
                         "in" parameters
parameter_test.adb:7:07: assignment to "in" mode
                         parameter not allowed
gnatmake: "parameter_test.adb" compilation error
```

Pass-By-Reference

- An access path is transmitted to the called subprogram:
  - usually the l-value of the actual parameter is passed on the stack.
- Also called *pass-by-sharing*.
- Advantage:
  - Passing process is efficient (no copying and no duplicated storage).
  - Can be used to implement all semantic modes (`in`, `out`, `inout`).
- Disadvantages:
  - Slower accesses to formal parameters (compared to pass-by-value).
  - Potentials for unwanted side effects (collisions).
  - Unwanted aliases (access broadened).

Pass-By-Reference (InOut Mode): Ada

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO; use Ada.Text_IO;

procedure Parameter_Test is

   type Employee is record
        Name: Unbounded_String;
        Hourly_Pay: Float;
   end record;

   procedure Fun_InOut(X: in out Employee) is
   begin
     X.Name := To_Unbounded_String("John Williams");
     X.Hourly_Pay := 20.0;
   end Fun_InOut;

   A: Employee :=
     (Name => To_Unbounded_String("Mark Brown"),
      Hourly_Pay => 15.0);
begin
   Fun_InOut(A);
   Put(To_String(A.Name) &
         " " &
         Float'Image(A.Hourly_Pay));
end Parameter_Test;
```

```
> gnat make parameter_test.adb
gcc-4.4 -c parameter_test.adb
gnatbind -x parameter_test.ali
gnatlink parameter_test.ali

> ./parameter_test
John Williams 2.00000E+01
```

Pass-By-Reference (InOut Mode) : C++

```
struct Employee {
    string  name_;
    float  hourly_pay_;
};

void funInOut(Employee &X)
{
    X.name = "John Williams";
    X.hourly_pay_ = 20.0;
}

int main(int argc, char** argv)
{
    Employee A = {"Mark Brown", 15.0};
    funInOut(A);
    cout << A.name << " " << A.hourly_pay_ << endl;
    return 0;
}
```

```
> g++ parameter_test.cc –o parameter_test

> ./parameter_test
John Williams 2.00000E+01
```

Pass-By-Reference (In/Out)

- *Pass by reference* can also be used to implement `in` mode:
  - C++: `void fun(const int &p1) { … };`
  - Disadvantages :
    - enforcing write protection in the callee is not easy.
    - accesses costs more (indirect addressing).
- *Pass by reference* can also be used to implement `out` mode:
  - C#: `void fun(out int x, out int y) { … };`
  - Ada: `procedure Fun_Out(X: out Employee) is`

Pass-by-Value-Result vs. Pass-by-Reference

```
procedure Parameter_Test is

   Y: Integer := 2;

   procedure Fun_InOut(X: in out Integer) is
   begin
      X := X + 1;
      X := X + Y;
   end Fun_InOut;

begin
   Fun_InOut(Y);
   Put(Y);
end Parameter_Test;
```

Pass-by-Value-Result ⇒ Y = ?

Pass-By-Reference ⇒ Y = ?

Aliasing

Pass-by-Name (InOut Mode)

- By textual substitution:
  - The actual parameter is textually substituted for the corresponding formal parameter in all its occurrences in the subprogram.
  - Potential for name conflicts.

- Introduced in Algol 60, but not part of any widely used language.

- Still used at preprocessing/compile time for:
  - macro substitution.
  - generic parameters for generic subprograms in C++ and Ada.

Parameter Passing Methods of Major Languages

- C:
  - *Pass-by-value.*
  - *Pass-by-reference* is simulated by using pointers as parameters.
- C++:
  - *Pass-by-reference* using a special pointer type called reference.
  - What is the difference between:
    - `void fun(int p1) { … };`
    - `void fun(const int &p1) { … };`
- Java:
  - All parameters are *passed by value*.
  - Object parameters are in effect *passed by reference*.

Parameter Passing Methods of Major Languages

- Ada:
  - Three semantics modes of parameter transmission: `in`, `out`, `in out`; `in` is the default mode:
    - Formal parameters declared `out` can be assigned:
      - can not be referenced in Ada 83 ⇒ awkward;
      - restriction removed in Ada 95.
    - those declared `in` can be referenced but not assigned;
    - `in out` parameters can be referenced and assigned
  - Scalar parameters are passed-by-copy.

    ⇒ `in out` implemented as *pass-by-value-result*.
  - Structured parameters are passed-by-reference.

    ⇒ `in out` implemented as *pass-by-reference*.

Parameter Passing Methods of Major Languages

- Fortran 95 is similar to Ada:
  - Parameters can be declared to be `in`, `out`, or `inout` mode.

- C#:
  - `in` mode is default, implemented as *pass-by-value*.
  - `out` mode is specified with the `out` modifier:
    - implemented as *pass-by-reference*.
  - `in out` mode is specified with the `ref` modifier:
    - implemented as *pass-by-reference*.

Parameter Passing Methods of Major Languages

- Perl:
  - all actual parameters are implicitly placed in a predefined array named `@_` whose elements are *aliases* for actual parameters.

- Python and Ruby:
  - *Pass-by-assignment* (all data values are objects, often immutable).
  - It is in effect semantically equivalent with *pass-by-reference*:
    - every variable stores a reference to an object.

    ⇒ the value of an actual parameter is a reference, that is assigned to the formal parameter.

Type Checking Parameters

- Considered very important for reliability.
- FORTRAN 77 and original C: none.
- Pascal, FORTRAN 90, Java, and Ada: it is always required.
- ANSI C and C++: choice is made by the user
  - Type checking avoided by using ellipsis (e.g. *printf*).
- Perl, JavaScript, and PHP do not require type checking.
- In Python and Ruby:
  - variables do not have types (objects do);
  - formal parameters are typeless;

  ⇒ parameter type checking is not possible.

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:
  - Many classes have overloaded constructors.
- C++, Java, C#, and Ada also allow users to write multiple versions of subprograms with the same name.

Overloaded Subprograms

- In Ada, the return type of an overloaded function can be used to disambiguate calls:
  - Possible because it does not allow mixed mode expressions.

```
A, B : Integer;
…
A := B + Fun(7);
```

- In C++, Java, and C# the return type is irrelevant to disambiguation of overloaded functions/methods:
  - Impossible because they allow mixed mode expressions.

Polymorphic & Generic Subprograms

- A **polymorphic** subprogram takes parameters of different types on different activations.
  - **ad hoc polymorphism =** the type of polymorphism provided by overloaded subprograms.
  - **parametric polymorphism =** the type of polimorphism provided by generic subprograms.
    - A **generic** subprogram is parameterized with type information, using a type expression that describes the type of the parameters.

- Example: generic function definition in C++:

```
template<class type> void swap(type& a, type&b)
{ type temp = a; a = b; b = temp; }
```

Polymorphic & Generic Subprograms

- The compiler takes care of generating instances of the subprogram:
  - In C++, they are instantiated implicitly, when the subprogram is named in a call or when its address is taken with the & operator:

```
int  u = 1;
int  v = 0;
swap(u,v);
⇒ void swap(int& a, int&b)
   {int temp = a; a = b; b = temp; }
```

  - In Ada, generic subprograms are instantiated explicitly:

```
procedure int_swap is
      new swap(type ⇒ Integer);
```

Generic Functions vs. Macros in C++

- Generic Function:

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

- “Equivalent” Macro:

```
#define max(a,b) ((a) > (b)) ? (a) : (b)
```

- Is there any difference?

Overloaded Operators

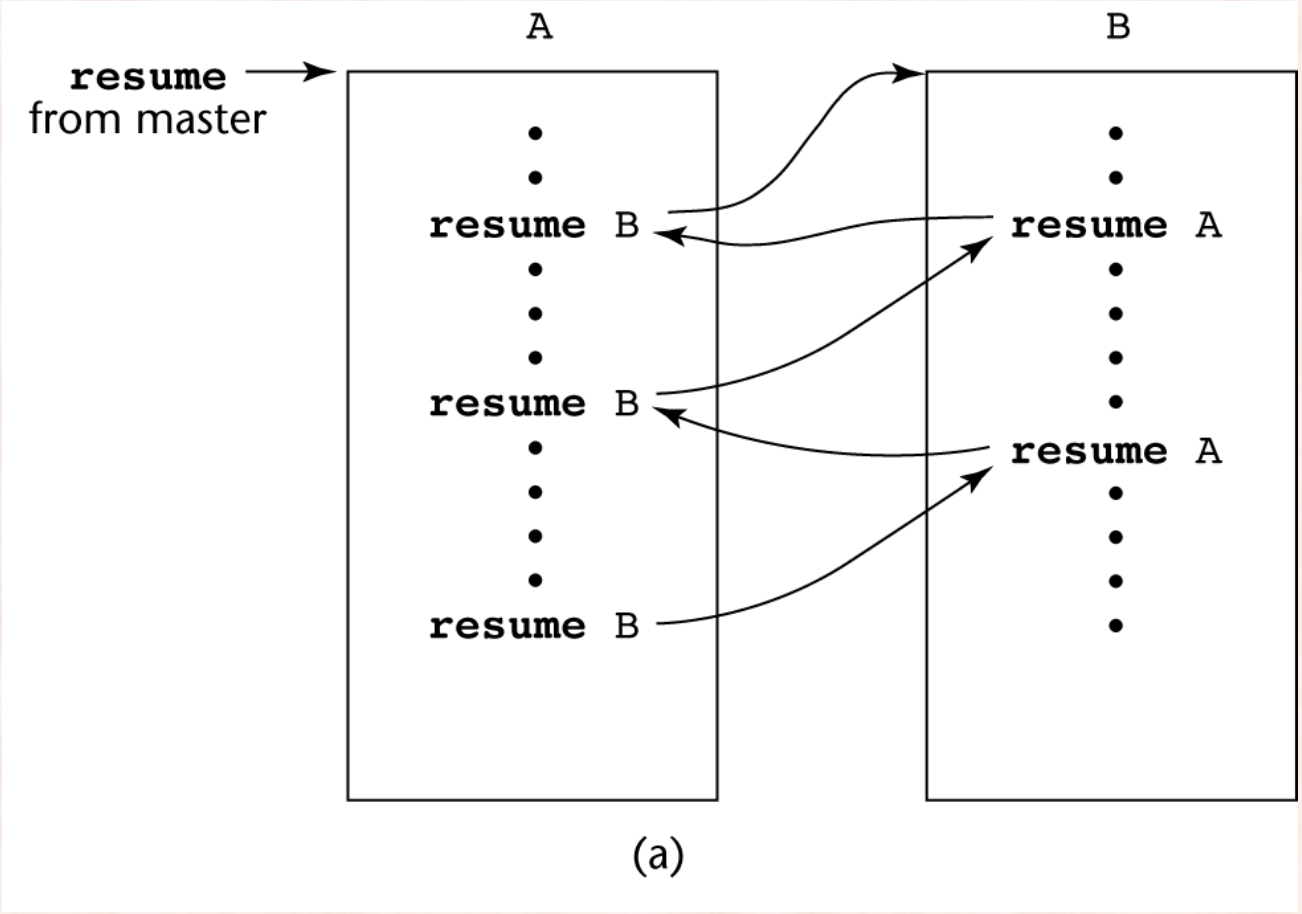
- Operators can be overloaded in Ada, C++, Python, and Ruby.
- An Ada example:

```
function "*" (A,B: in Vec_Type): return Integer
  is
  Sum: Integer := 0;
  begin
  for Index in A'range loop
   Sum := Sum + A(Index) * B(Index)
  end loop
  return sum;
end "*";
…
c = a * b; -- a, b are of type Vec_Type
           -- c is of type Integer
```

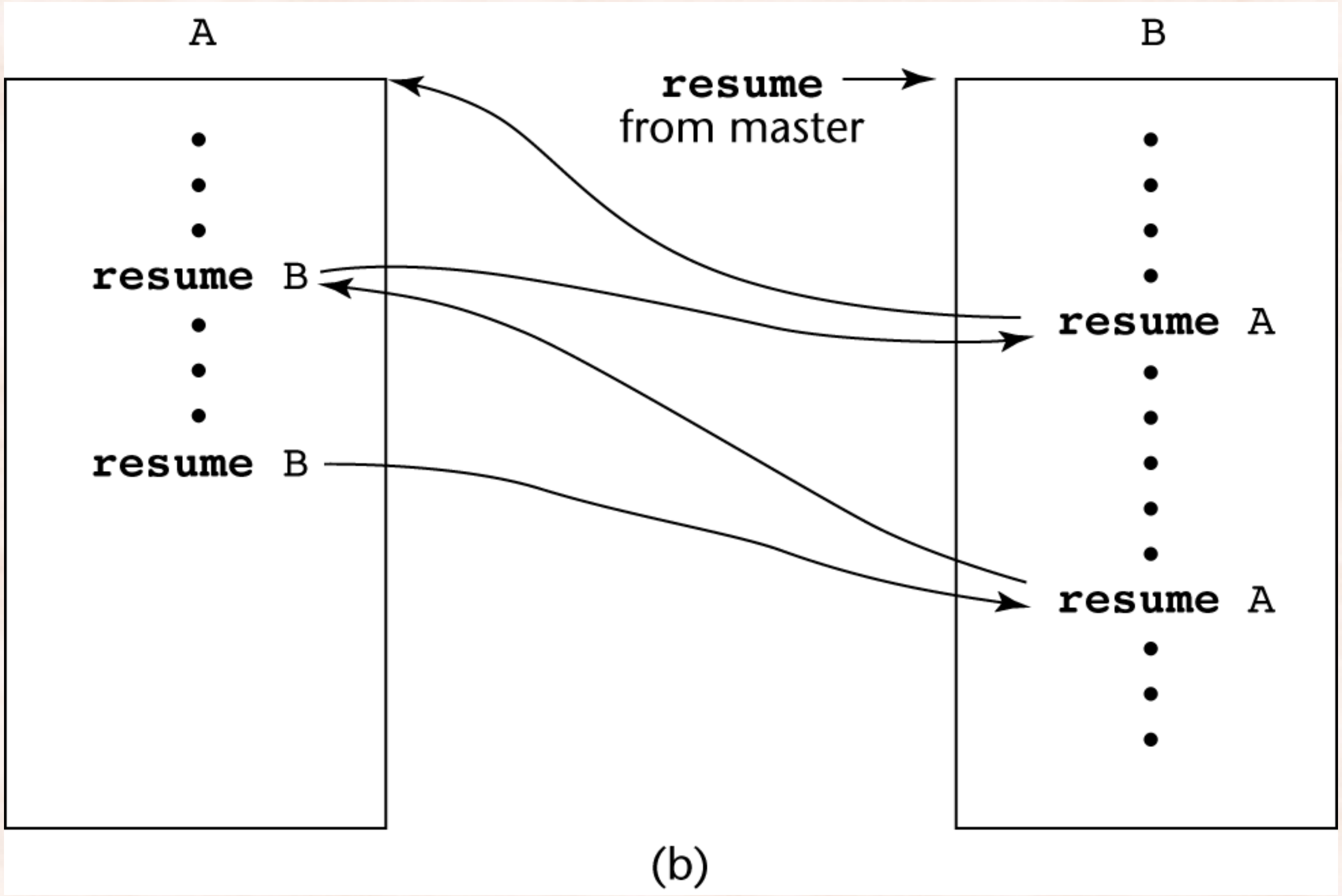
Coroutines

- A **coroutine** is a subprogram that has multiple entries and controls them itself.
- **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;
  - Subsequent resumes 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.

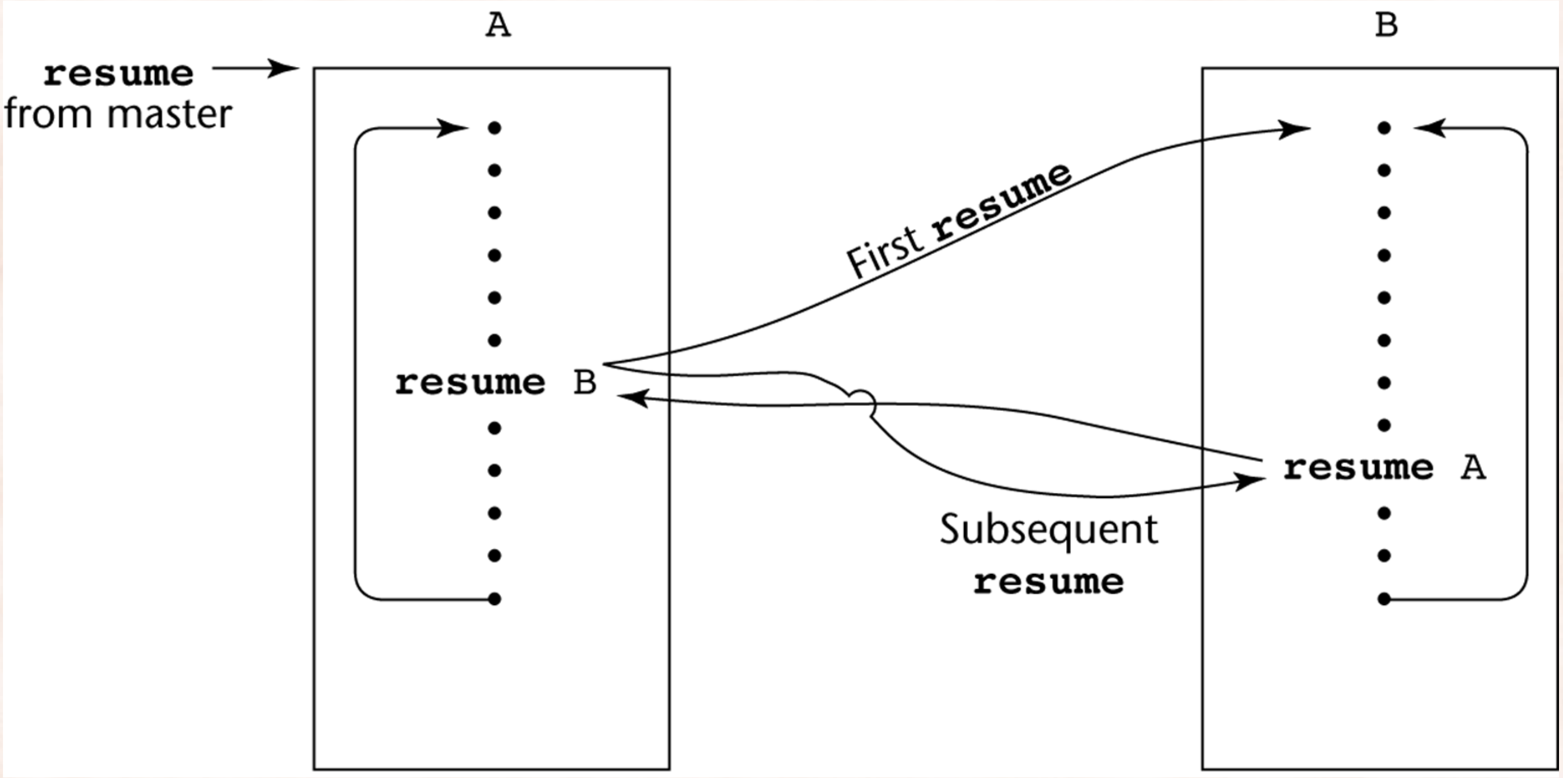
Coroutines Illustrated: Possible Execution Controls



Coroutines Illustrated: Possible Execution Controls

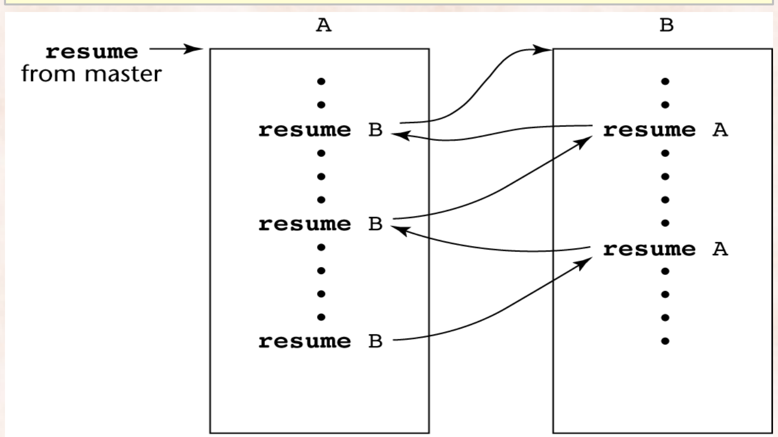


Coroutines Illustrated: Possible Execution Controls with Loops



Simulating Coroutines Using Generators in Python

```
def master():
    gens = [None, None]
    gens[0] = funA(gens)
    gens[1] = funB(gens)
    gen = gens[0]    # start with A
    try:
        while True:
            gen = gen.next()
    except StopIteration:
        None
```



(a)

```
def funA(gens):
    print "In A, right before first resume B."
    yield gens[1]
    print "In A, right before second resume B."
    yield gens[1]
    print "In A, right before third resume B."
    yield gens[1]


def funB(gens):
    print "In B, right before first resume A."
    yield gens[0]
    print "In B, right before second resume A."
    yield gens[0]
    print "In B, right at the end."
```