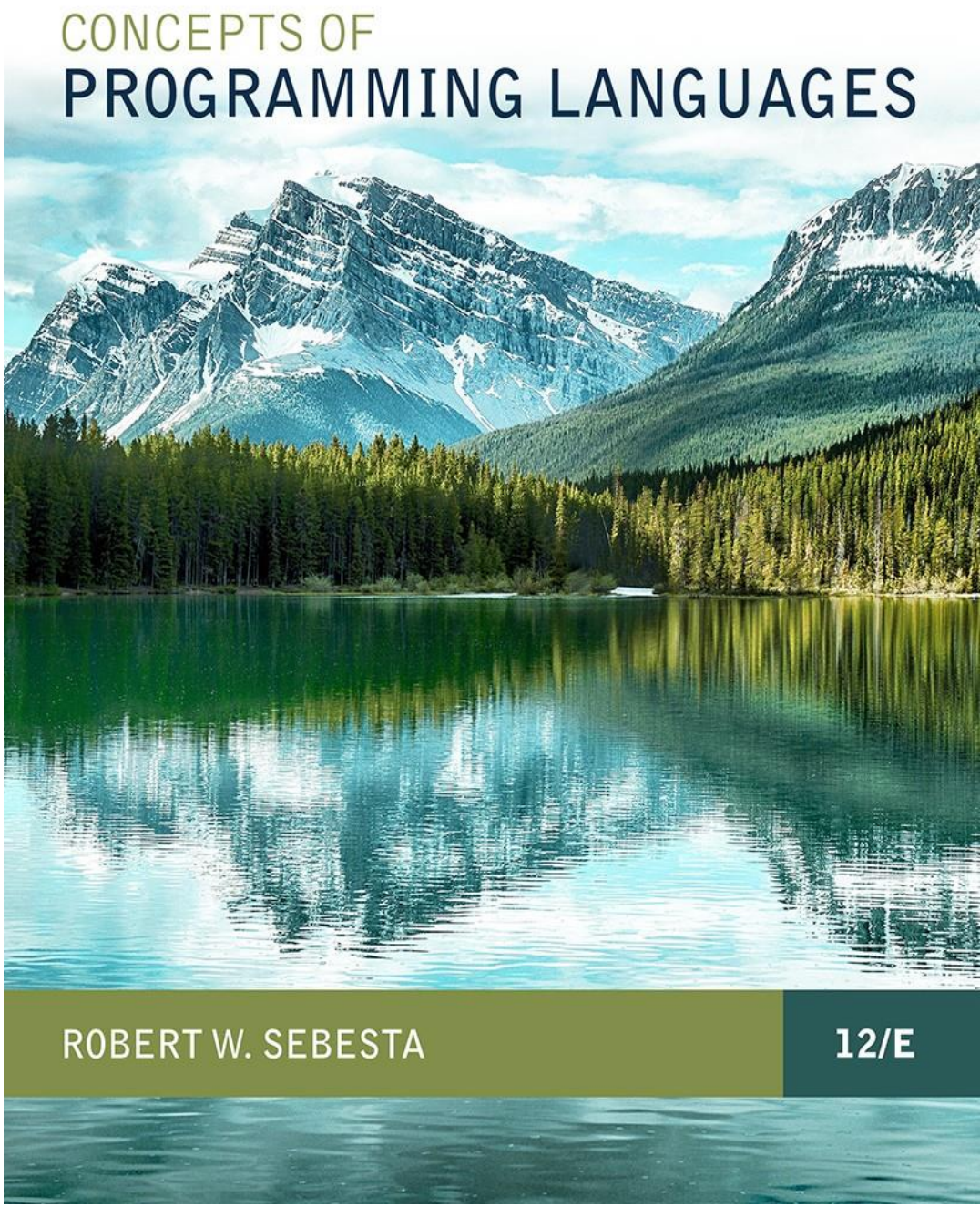
Chapter 9

Subprograms



ISBN 0-321-49362-1

Chapter 9 Topics

- 브프로그램의 원리
- 부프로그램의 설계 고려사항
- 지역 참조 환경
- 매개변수-전달 방법
- 부프로그램 매개변수
- 간접 부프로그램 호출
- 중복 부프로그램
- 포괄형 부프로그램
- 사용자-정의 중복 연산자
- 클로저
- 코루틴

일반적인 부프로그램의 특성

- 각 부프로그램은 단일 진입점을 가짐
- 호출 프로그램은 피호출 프로그램의 실행 중에는 중단됨
- 피호출 프로그램 실행이 끝났을 때, 제어는 항상 호출 프로그램에게 돌아감

기본 정의(1)

- 부프로그램 정의 : 부프로그램 추상화의 동작과 인터페이스를 서술
  - Python에서 함수 정의는 실행가능; 다른 언어에서는 실행불가
  - Ruby에서 함수 정의는 클래스 내부나 외부에 올 수 있다. 외부에서 정의되면 루트 객체 Object의 매소드이며 객체 없이 호출될 수 있음
  - Lua에서는 모든 함수가 무명이다.
- 부프로그램 호출 : 부프로그램 수행의 명시적 요청
- 부프로그램 헤더: 이름, 종류, 형식 매개변수를 포함
- 부프로그램의 매개변수 프로파일(시그니쳐) : 형식매개변수의 개수, 순서, 타입
- 부프로그램의 프로토콜 : 매개변수 프로파일(함수이면 반환 타입도 포함)

Python의 함수 특징

def Execute at Runtime

The **def** is an executable statement. When it runs, it creates a new function object and assigns it to a name. Because it's a statement, a **def** can appear anywhere a statement can even nested in other statements:

```
# function.py
def func():
    print('func()')
    def func1():
        print('func1')
        def func2():
            print('func2')
        func2()
    func1()

func()
```

Output should look like this:

```
$ python function.py
func()
func1
func2
```

기본 정의(2)

- C와 C++에서 함수선언은 prototype이라 부름
- 부프로그램의 선언: 프로토콜만 제공(body는 불포함)
- 형식매개변수(부프로그램 헤더에 사용)
- 실매개변수(호출문에 사용)

실/형식 매개변수 대응방법

- 위치 매개변수(positional parameter)
  - 실매개변수를 형식매개변수로 바인딩은 위치에 의해 동작 : 첫 실매개변수는 첫 형식매개변수에 바인드, …
  - 효율적이고 안전(매개변수 리스트가 짧은 경우)

- 키워드 매개변수(keyword parameter)
  - 실매개변수가 바인딩될 형식매개변수의 이름을 명세

    (ex) Python에서

    ```
    sumer(length=my_length, list=my_array, sum=my_sum)
    ```
  - 장점 : 매개변수가 무순서, 매개변수 대응 오류 회피
  - 단점 : 사용자는 형식매개변수의 이름을 알아야함

    (ex) Python은 위치매개변수도 허용(혼용 가능)

    ```
    sumer(my_length, sum=my_sum, list=my_array)
    ```

형식매개변수 디폴트 값(1)

- C++, Python, Ruby, PHP 등의 언어에서 형식매개변수는 디폴트 값 가능(실매개변수 값이 전달되지 않는 경우)

(ex) C++ : 디폴트 매개변수는 매개변수가 위치적으로 대응하기 때문에 마지막에 나타나야 함(키워드 매개변수 지원 않음)

```
float compute_pay(float income, float tax_rate,
                                  int exemptions=1)
pay = compute_pay(20000.0, 0.15);
```

(ex) Python :

```
def compute_pay(income, exemptions=1, tax_rate)
pay = compute_pay(20000.0, tax_rate=0.15)
```

형식매개변수 디폴트 값(2)

- 가변 개수의 매개변수
  - C# 매소드는 타입이 동일한 가변 개수의 매개변수 허용 – 매소드는 params 수정자를 형식매개변수에 명세; 호출은 배열이나 표현식의 리스트를 전달

(Ex)
```
public void DisplayList(params int[] list){
    foreach (int next in list){
        Console.WriteLine("Next value {0}", next);
    }
}
Myclass myObject = new Myclass();
int[] myList = new int[6] {2,4,6,8,10,12};
    . . .
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3*x-1, 17);
```

  - In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk

프로시저와 함수

- 부프로그램의 두가지 유형
  - **프로시저** : 매개변수화된 계산을 정의하는 문장들의 집합
    - 리턴값은 없음
    - 프로시저 지원언어 : Ada(프로시저), Fortran(서브루틴)
  - **함수** : 구조적으로 프로시저와 유사, 의미적으로 수학 함수가 모델이다.
    - 리턴값이 있음
    - 실제적으로 프로그램 함수는 부작용을 가짐
    - 부작용이 없는 함수는 리턴값이 유일한 결과값

부프로그램의 설계 고려사항

- 지역변수는 정적 혹은 동적?
- 부프로그램 정의 내에 다른 부프로그램 정의 포함가능?
- 어떤 매개변수 전달방법을 제공?
- 매개변수 타입 검사?
- 만약 부프로그램이 매개변수로 전달되고 중첩 정의 가능하면, 부프로그램의 참조환경은?
- 부프로그램의 중복정의?
- 부프로그램의 포괄형?
- 중첩 부프로그램을 허용한다면 클로저 지원 여부?

지역 참조 환경

- 지역변수가 스택-동적이라면
  - 장점 : 유연성
    - Recursion 지원
    - 부프로그램간에 기억장소 공유
  - 단점
    - 할당/회수, 초기화 타임
    - Indirect addressing
    - 부프로그램은 과거민감 불가
- 지역변수가 정적이라면
  - 스택-동적 지역변수의 장단점의 반대

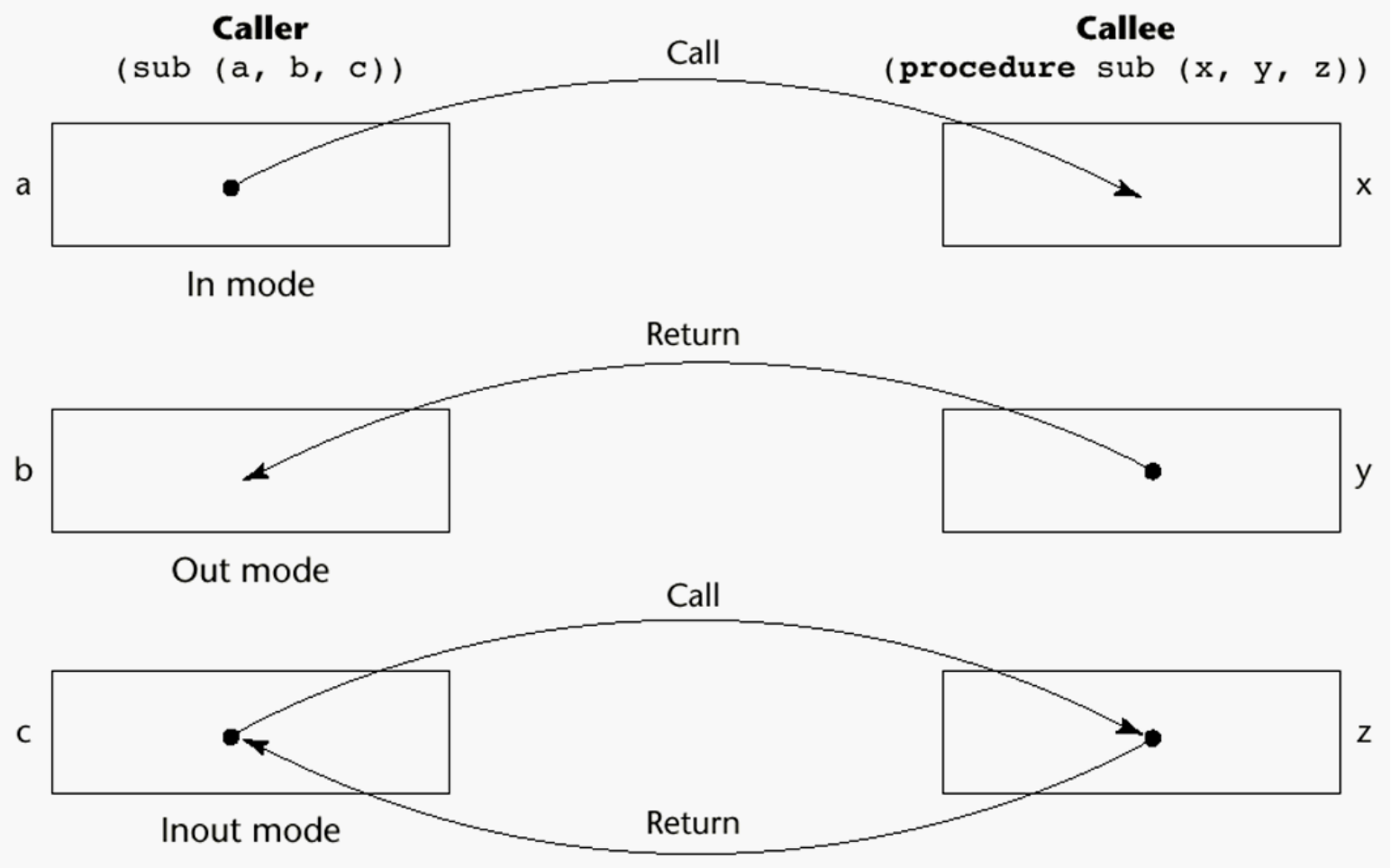
지역 참조 환경 : Examples

- 대부분 기존 언어에서 지역변수는 스택-동적
- C-기반 언어에서, **`static`** 선언을 하지 않으면 스택-동적임
- C++, Java, Python, C# 의 매소드는 단지 스택-동적 지역변수만 가짐

매개변수 전달의 의미적 모델

- 입력모드(In mode)
- 출력모드(Out mode)
- 입출력모드(Inout mode)

매개변수 전달의 의미적 모델



전달의 개념적인 모델

- 값이 물리적으로 복사
- 값의 접근경로가 이동
  (접근경로는 단순 포인터나 참조)

매개변수 전달의 구현 모델

- 값-전달
- 결과-전달
- 값-결과-전달
- 참조-전달
- 이름-전달

값-전달(Pass-by-Value):In Mode

- 실매개변수의 값은 해당 형식매개변수를 초기화하기 위해 사용된다.
  - 보통 복사로 구현
  - 실매개변수 값의 접근경로 전달로 구현할 수 있지만 비추천 (값이 쓰기-보호 셀에 있게 하는 것은 쉽지 않음)
  - 단점 (if by physical move): 추가 기억장소 필요, 매개변수가 많을 때 실제 이동 비용 상승
  - 단점 (if by access path method): 부프로그램에서 쓰기 보호, 간접어드레싱으로 접근비용

결과-전달(Pass-by-Result):Out Mode

- 부프로그램에 전달되는 값은 없음; 해당 형식매개변수는 지역변수처럼 사용;
- 제어가 호출자에게 반환될 때 형식매개변수 값은 호출자의 실매개변수(반드시 변수여야 함)로 전달
  - 별도의 기억공간과 copy 연산 필요

- 잠재적인 문제점: 실매개변수의 충돌(실매개변수에 배정되는 순서의 문제점)

  (예제)

결과-전달(Pass-by-Result):Out Mode (continued)

- `sub(p1, p1);` 두 형식매개변수가 다른 이름을 갖는다면, 대응되는 실매개변수에 나중에 배정되는 것이 `p1`의 값이 됨

```
void Fixer(out int x, out int y)  {
   x=17;
   y=35;
}
   . . .
f.Fixer(out a, out a);
```

- `sub(list[sub], sub);` list[sub]의 주소 평가는 호출시 혹은 복귀시?

```
void DoIt(out int x, out int index) {
   x=17;
   index=42;
}
   . . .
sub=21;
f.DoIt(list[sub], sub);
```

값-결과-전달(Pass-by-Value-Result) : inout Mode

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
  - Those of pass-by-result
  - Those of pass-by-value

참조-전달(Pass-by-Reference) : Inout Mode

- Pass an access path
- Also called pass-by-sharing
- Advantage : Passing process is efficient
  (no copying and no duplicated storage)
- Disadvantages
  - Slower accesses (compared to pass-by-value) to formal parameters
  - Potentials for unwanted side effects (collisions)
  - Unwanted aliases (access broadened)

  (예제)

참조-전달(Pass-by-Reference) : Inout Mode(continued)

- Unwanted aliases (access broadened)

```
void fun(int &first, int &second)
{   . . .    }
```

```
fun(total, total);      // first와 second는 별칭
fun(list[i], list[j];   // i와 j가 같다면 first와
                           second는 별칭
fun1(list[i], list);    // list의 모든 원소와 한 원소는 별칭
```

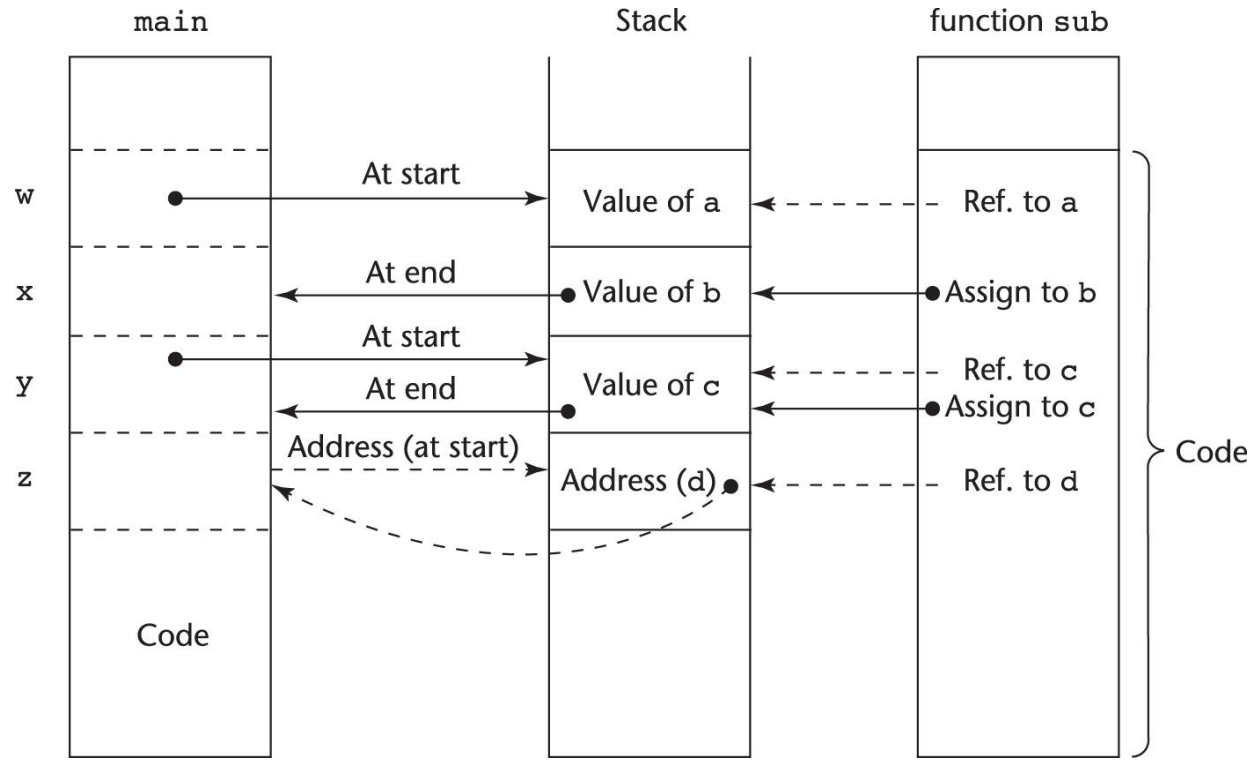
이름-전달(Pass-by-Name) : Inout Mode

- 문자 그대로 대체
- 호출시 접근방법에 바인드 되나, 값이나 주소에 실제 바인딩은 형식매개변수에 값이 배정되거나 참조될 때까지 연기
- 지연 바인딩(late binding)의 유연성
- 구현을 위해서는 호출자의 참조환경이 매개변수로 전달되어야 실매개변수의 주소가 계산될 수 있음
- 구현이 어렵고 비능률적, 복잡성으로 판독성과 신뢰성 저하

매개변수-전달 방법의 구현

- 대부분 언어에서 매개변수 전달이 실행시간-스택을 통해 수행
- 참조-전달이 구현하기 가장 단순한 방법임; 단지 주소만 스택에 위치

매개변수-전달 방법의 구현 (continued)



Function header: `void sub(int a, int b, int c, int d)`
Function call in main: `sub(w, x, y, z)`
(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)

일반적인 언어의 매개변수 전달 방법

- C
  - 값-전달
  - 포인터 타입 매개변수로 참조-전달 효과
- C++
  - 참조 타입이라는 특수한 포인터 타입을 사용하여 참조-전달
  - 참조 매개변수가 상수로 정의 가능
    (ex) void fun(const int &p1, int p2, int &p3);
- Java
  - 모든 비객체 매개변수는 값-전달
  - 객체 매개변수는 참조-전달

일반적인 언어의 매개변수 전달 방법(continued)

- Fortran 95+
  - in, out, or inout mode
- C#
  - 디폴트 방법 : 값-전달
  - 참조-전달은 형식매개변수와 대응되는 실매개변수에 `ref`를 선행시킴으로 지정

    (ex) void sumer(ref int oldSum, int newOne) { . . . }

    . . .

    sumer(ref sum, newValue);

일반적인 언어의 매개변수 전달 방법(continued)

- Python과 Ruby: 배정-전달(pass-by-assignment); 모든 데이터 값은 객체이므로 모든 변수는 객체에 대한 참조; 실매개변수가 형식매개변수에 배정

(ex) Python에서

```
>>> def h(t):
        t=(1,2,3)
>>> a=(5,6,7)
>>> h(a)
>>> print(a)
(5,6,7)
```

```
>>> def g(t):
        t[1]=10
>>> a=[1,2,3]
>>> g(a)
>>> print(a)
[1,10,3]
```

매개변수의 타입 검사

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: it is always required
- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

매개변수로서 다차원 배열

- 만약 다차원 배열이 부프로그램에 전달되고 메인 프로그램과 분리하여 컴파일 된다면, 컴파일러는 사상함수를 구성하기 위해 그 배열의 선언된 크기를 알아야 한다.
- 주소(mat[i, j])=주소(mat[0, 0]) +
  i * 열의 수 + j (원소크기=1)

매개변수로서 다차원 배열 : C and C++

- 프로그래머는 실매개변수의 첫 첨자를 제외한 모든 배열의 크기를 선언해야 함

```
void fun(int matrix[][10]) {  . . . }
void main() {
   int mat[5][10];
      . . .
   fun(mat);
      . . .   }
```

- 유연한 부프로그램 작성을 불허
- 해결책은?

매개변수로서 다차원 배열 : C and C++

- 해결책 : 배열은 포인터로 전달, 배열의 실제 차원은 매개변수로 전달; 함수는 배열의 원소가 참조될 때마다 포인터 연산을 사용하여 사용자-정의 기억장소-사상 함수를 평가한다.

```
void fun(float *mat_ptr, int num_rows, int num_cols)  {
        . . .
      *(mat_ptr + (row * num_cols) + col) = x;
```

- 매크로 정의

```
#define mat_ptr(r, c)   *(mat_ptr + ((r) * (num_cols) + (c)))
        . . .
        mat_ptr(row, col) = x;
```

매개변수로서 다차원 배열 : Java and C#

- 배열은 객체; 배열은 모두 1차원 이지만 원소가 배열이 될 수 있음
- 각 배열은 배열 객체가 생성될 때 배열의 길이로 설정된 이름 상수(`length` in Java, `Length` in C#) 를 상속받음

(Ex)
```
float  sumer(float mat[ ][ ]) {
  float sum=0.0f;
  for (int row = 0; row < mat.length; row++) {
     for (int col = 0; col < mat[row].length; col++) {
          sum += mat[row][col];       }
     }
  return sum;
}
```

매개변수전달의 설계 고려사항

- 두 가지 중요한 고려사항
  - Efficiency
  - One-way or two-way data transfer
- 그러나 위의 고려사항은 다음과 같이 상충된다.
  - 좋은 프로그래밍은 변수의 제한된 접근(가능한 한 단-방향 데이터 이동)을 권장
  - 그러나 참조 전달은 큰 사이즈의 구조를 전달하기에 더 효율적

부프로그램 매개변수

- 부프로그램 이름이 다른 부프로그램의 매개변수로 전달된다면 아주 편리
- Issues:
  1. 매개변수의 타입 검사문제
  2. 매개변수로 전달된 부프로그램을 실행하기 위해 어떤 참조 환경이 사용되어야 하는가?
     (중첩 부프로그램 허용 언어에서)

부프로그램 매개변수: 참조 환경

- 피상 바인딩(Shallow binding): 전달된 부프로그램을 실행시키는 호출문의 환경
  - 동적-영역 언어에 적당
- 심층 바인딩(Deep binding): 전달된 부프로그램의 정의 환경
  - 정적-영역 언어에 적당
- 특이 바인딩(Ad hoc binding): 부프로그램을 실 매개변수로 전달한 호출문의 환경

(Ex) P.450

```
function sub1() {                          //  JavaScript
   var x;
   function sub2() {
      alert(x);  // create 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();
   };
```

sub2의 실행을 보면
- 피상 바인딩에서 실행의 참조환경 : sub4의 환경,
  sub2에서 x의 참조는 sub4의 지역변수 x (4)
- 심층 바인딩에서 sub2 실행의 참조환경:sub1의 환경,
  sub2에서 x의 참조는 sub1의 지역변수 x (1)
- 특이 바인딩에서 sub2 실행의 참조환경:sub3의 환경,
  sub2에서 x의 참조는 sub3의 지역변수 x (3)

부프로그램 간접 호출

- 부프로그램이 간접적으로 호출되어야 하는 경우는 호출될 특정 부프로그램이 실행될 때 까지 알려지지 않을 때임
- C와 C++에서는,
  - 포인터나 참조를 통해서 함수 호출 가능
  - 함수는 매개변수로 전달불가, 함수 포인터는 가능

```
int  myfun2(int, int);
int  (*pfun2) (int, int) = myfun2;
//    pfun2 = myfun2;
…
(*pfun2)(first, second);
//    pfun2(first, second);
```

부프로그램 간접 호출 (continued)

- C#에서, 메소드 포인터는 위임자(delegate)라 부르는 객체로 구현
- 위임자의 선언:

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

: 이러한 위임자 타입, 이름이 Change는 **int**형 매개변수를 갖고 **int** 값을 반환하는 메소드로 사례화될 수 있다.

- 메소드 정의: `static int fun1(int x) { … }`
- 객체화: `Change chgfun1 = new Change(fun1);`
- 위임자를 통한 함수호출: `chgfun1(12);`
- 다중 위임자(Multicast delegate) : 1개 이상의 주소를 저장하는 위임자

```
Change chgfun1 += fun2;
```

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

중복 부프로그램

- 중복 부프로그램은 같은 참조 환경에서 부프로그램과 이름이 같은 다른 부프로그램이다.
  - 모든 중복 부프로그램은 각기 고유한 프로토콜을 갖는다.
- C++, Java, C#은 미리 정의된 중복 부프로그램을 포함한다.
- Java, C++, C# 은 같은 이름을 갖는 여러 개의 부프로그램을 작성할 수 있다. (가장 일반적인 사용자-정의 중복 메소드는 생성자임)
- 디폴트 매개변수를 갖는 중복 부프로그램은 모호한 호출

(ex)
```
void fun(float b=0.0);
void fun( );
. . .
fun( );      // 모호하며 컴파일오류
```

포괄형 부프로그램

- 포괄형 혹은 다형 부프로그램은 다른 타입의 매개변수를 가지고 다르게 활성화한다.
- 중복정의된 부프로그램은 특이 다형성(*ad hoc polymorphism)*이라는 특수한 종류의 다형성 제공
- *부타입 다형성(subtype polymorphism)*은 타입 T의 변수는 타입 T 또는 T에서 파생한 타입의 객체를 접근할 수 있음을 의미 (OOP languages)
- 부프로그램의 매개변수의 타입을 포괄형 매개변수로 사용하는 부프로그램은 매개변수 다형성(*parametric polymorphism*)을 제공
  - A cheap compile-time substitute for dynamic binding

포괄형 부프로그램 (continued)

- C++
  - 포괄형 함수의 버전은 함수가 호출되었을 때 묵시적으로 생성된다.
  - 포괄형 함수는 **template** 포괄형타입리스트 절로 시작하며, 포괄형 타입은 typename 혹은 class가 될 수 있음

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

```
int a, b, c;
char d, e, f;
c = max(a, b);
f = max(d, e);
```

포괄형 부프로그램 (continued)

- Java 5.0
  - C++ 포괄형과의 차이점:

  1. Java 5.0의 포괄형 매개변수는 클래스여야 함
  2. 포괄형 메소드는 여러 번 사례화 될 수 있을지라도 단지 하나의 코드만이 구성됨
  3. 포괄형 매개변수로서 포괄형 메소드에 전달될 수 있는 클래스의 범위에 대한 제한을 명세할 수 있음
  4. 포괄형 매개변수의 만능(Wildcard) 타입 지원

포괄형 부프로그램 (continued)

- Java 5.0 (continued)

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

  - 매개변수는 포괄형 원소들의 배열임(`T` 는 타입의 이름)
  - 함수 호출: `doIt<String>(myList);`

포괄형 매개변수에 범위를 갖는 **doIt**의 다음 버젼:

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

포괄형 타입은 `Comparable` 인터페이스를 구현하는 클래스이어야 함

포괄형 부프로그램 (continued)

- Java 5.0 (continued)
  - 만능(Wildcard) 타입

    Collection<?> : 컬렉션 클래스에 대한 만능 타입

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

    → 어떠한 컬렉션 클래스에도 원소를 출력

포괄형 부프로그램 (continued)

- C# 2005
  - Java 5.0의 포괄형 메소드와 유사
  - 한가지 차이점: 호출에서 실 매개변수 타입은 컴파일러가 미지정 타입을 유추할 수 있다면 생략가능
    - 다른 한가지 : C# 2005은 만능 타입을 지원하지 않음

사용자-정의 중복 연산자

- Ada, C++, Python, and Ruby에서 연산자 중복 정의 가능

- Python의 예

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

사용: `x + y`를 계산하려면, `x.__add__(y)`

- C++의 예

```
Complex operator +(Complex &second) {
    return Complex(real + second.real,  imag + second.imag);
}
```

클로저(Closure)

- *closure*는 부프로그램이며 부프로그램이 정의된 참조환경
  - 만약 부프로그램이 프로그램의 임의의 장소로부터 호출될 수 있다면 참조환경은 필요하다.
  - 정적영역 언어가 중첩부프로그램을 허용하지 않는다면 클로저는 필요 없음
  - 부프로그램이 중첩 영역내 변수를 참조할 수 있고 어느 위치에서든 호출가능 하다면 클로저는 필요
  - 클로저는 함수 안에 무명의 함수를 정의하는 것
  - 외부 함수를 벗어날 때 내부의 무명함수를 참조할 수 있게 해줌
  - 지역변수를 의미하는 클로저는 영역을 벗어나더라도 내부 무명함수를 검색할 여지를 남겨두어서 필요하지 않을 때까지 변수를 연결해줌

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)

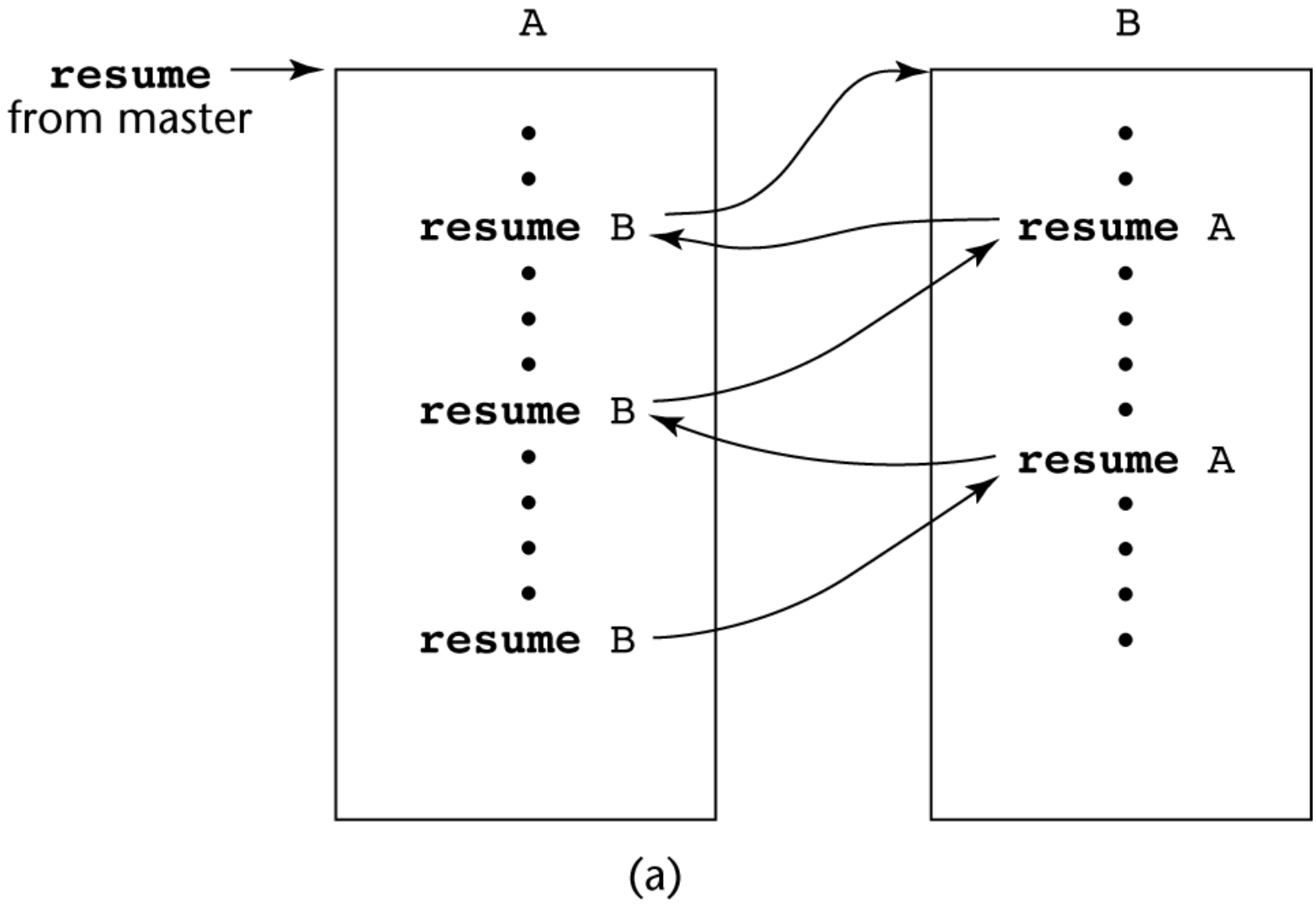
- C#
  - We can write the same closure in C# using a nested anonymous delegate
  - `Func<int, int>` (the return type) specifies a delegate that takes an `int` as a parameter and returns and `int`

```
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

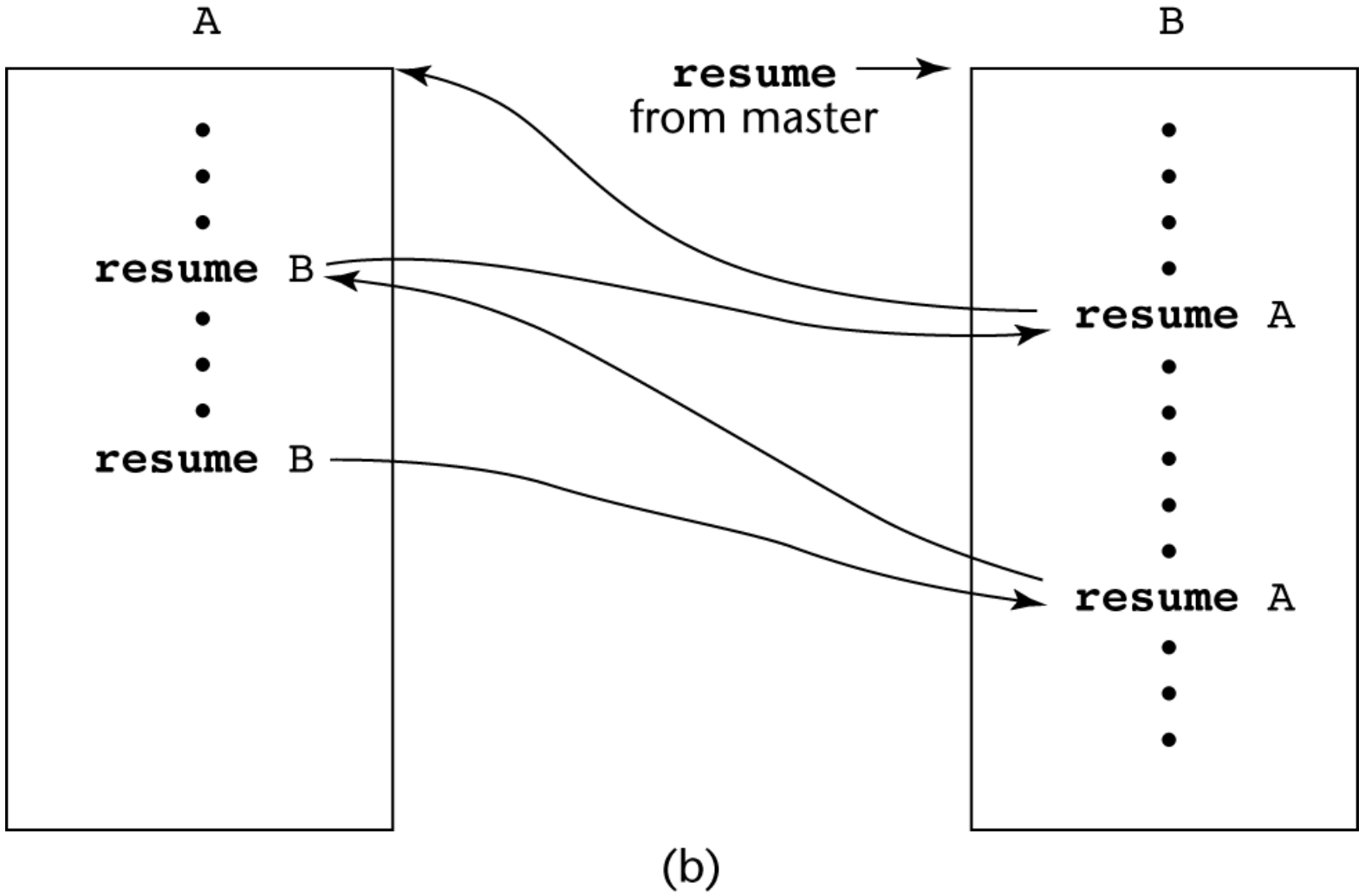
- A *coroutine* is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called *symmetric control:* caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent 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



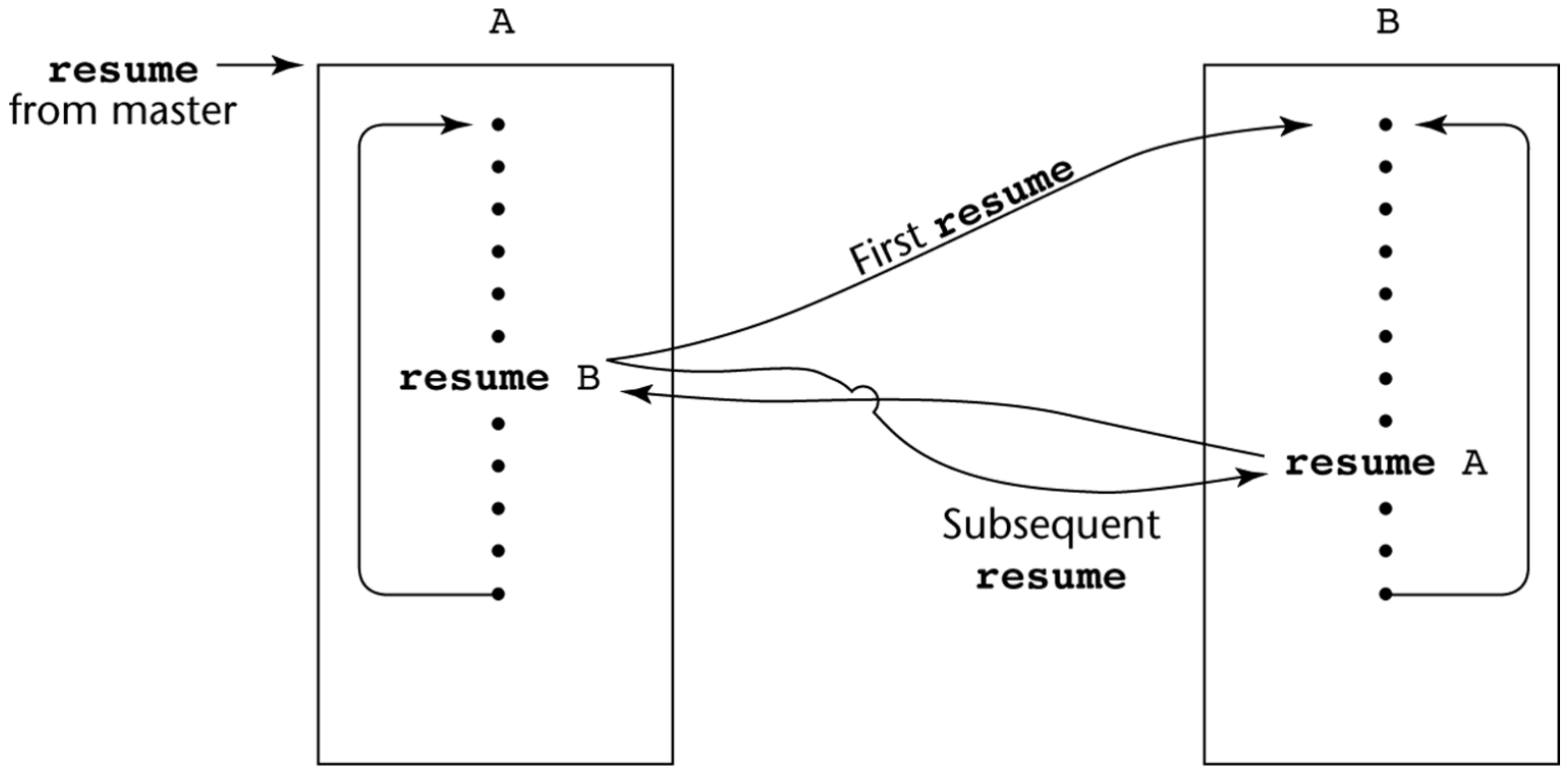
(a)

Coroutines Illustrated: Possible Execution Controls



(b)

Coroutines Illustrated: Possible Execution Controls with Loops



Summary

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A closure is a subprogram and its ref. environment
- A coroutine is a special subprogram with multiple entries