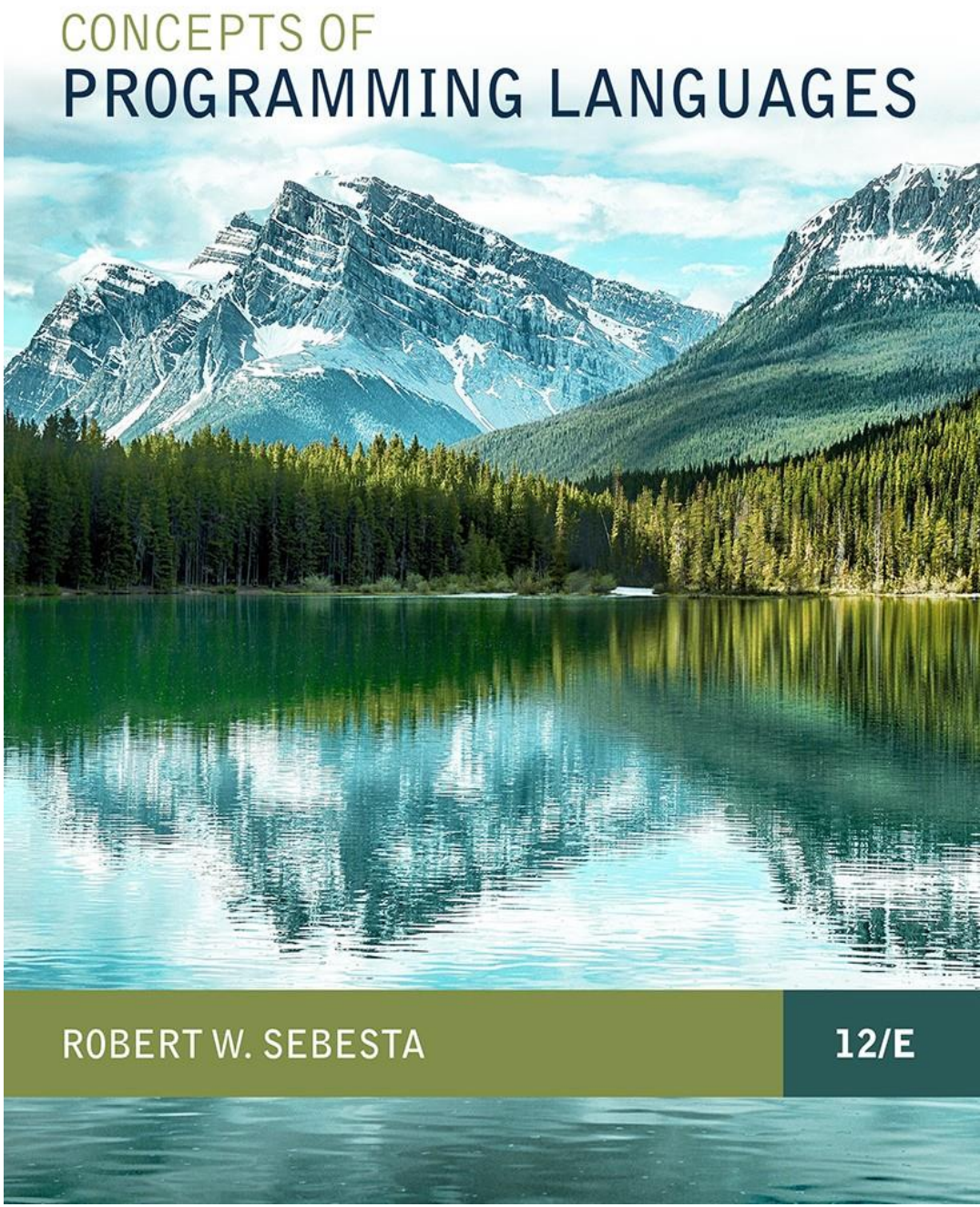
Chapter 10

Implementing Subprograms



ISBN 0- 0-321-49362-1

Chapter 10 Topics

- 호출과 복귀의 일반적 의미
- “단순” 부프로그램의 구현
- 스택-동적 지역 변수를 갖는 부프로그램의 구현
- 중첩 부프로그램
- 블록
- 동적-영역 규칙의 구현

호출과 복귀의 일반적 의미

- 부프로그램 호출과 복귀 연산을 통틀어 부프로그램 연결(*subprogram linkage*)이라 함

- 부프로그램 호출의 일반적 의미:
  - 매개변수 전달 방법
  - 지역변수를 위한 스택-동적 할당
  - 호출 프로그램의 실행 상태를 저장
  - 복귀를 위한 제어의 전달 확인
  - 중첩 부프로그램을 지원한다면, 비지역변수의 접근이 가능토록 준비

호출과 복귀의 일반적 의미

- 부프로그램 복귀의 일반적 의미:
  - 출력모드 또는 입출력 모드이면서 복사전달 매개변수를 가지면 그 지역 값을 실매개변수로 이동
  - 스택-동적 지역변수를 해제
  - Caller(호출 프로그램)의 실행 상태 복원
  - Caller로 제어 반환

“단순” 부프로그램의 구현

- “단순”: 중첩이 안되고 지역변수는 정적

- 호출의 의미:

  1. **caller**의 실행 상태 저장
  2. 매개변수 전달
  3. 복귀주소를 **called**(피호출 프로그램)에 전달
  4. 제어를 **called**에 전달

“단순” 부프로그램의 구현 (continued)

- 복귀의 의미:
  1. 값-결과-전달 매개변수나 출력-모드 매개변수가 사용된다면, 매개변수의 값은 대응되는 실 매개변수로 이동
  2. 부프로그램이 함수라면, 함수 값은 **caller**가 접근가능한 장소로 이동
  3. **caller**의 실행상태 복원
  4. 제어는 **caller**에게 돌아감

- 호출과 복귀에 필요한 기억장소:
  - **caller**의 상태정보, 매개변수, 복귀주소, 함수의 반환 값, 부프로그램의 임시 기억 장소

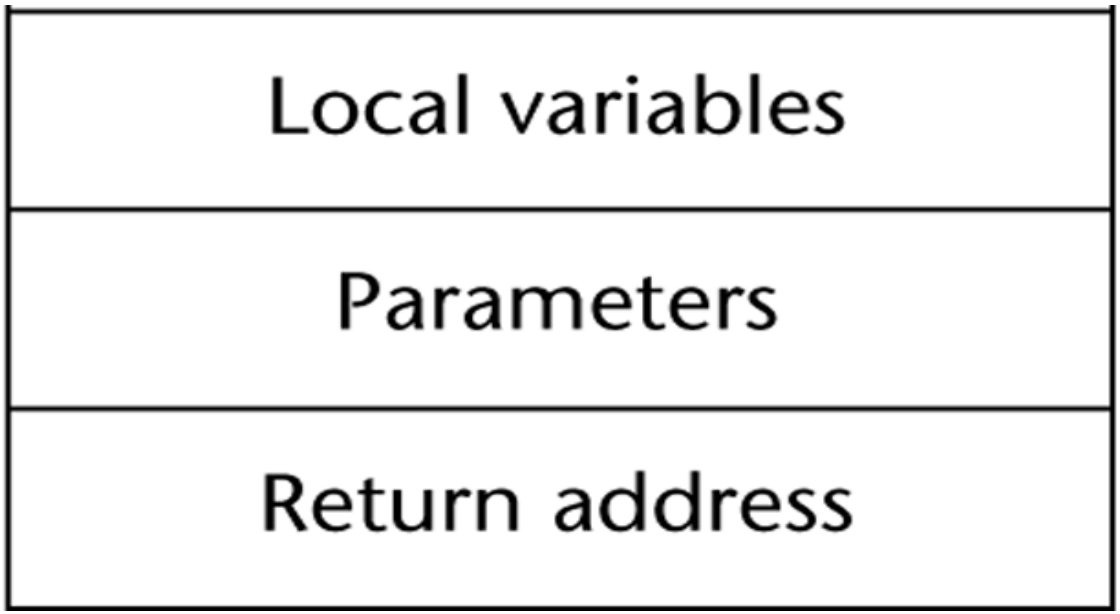
“단순” 부프로그램의 구현 (continued)

- 별도의 두 부분:
  - 코드 부분(부프로그램 코드)
  - 비코드 부분(지역변수와 데이터들)

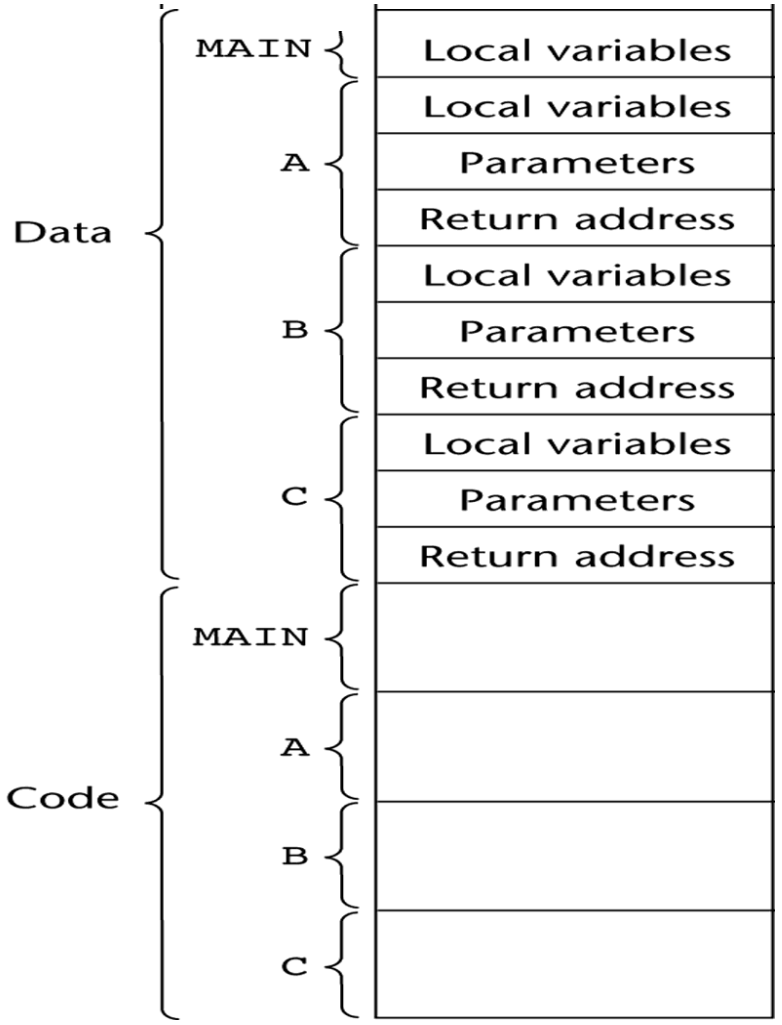
- 실행중인 부프로그램의 비코드 부분의 형식 (layout)
  → 활성화 레코드(*activation record)*

- 활성화 레코드 사례(*activation record instance : ARI)* 는 활성화 레코드의 구체적인 예

“단순” 부프로그램의 활성화 레코드



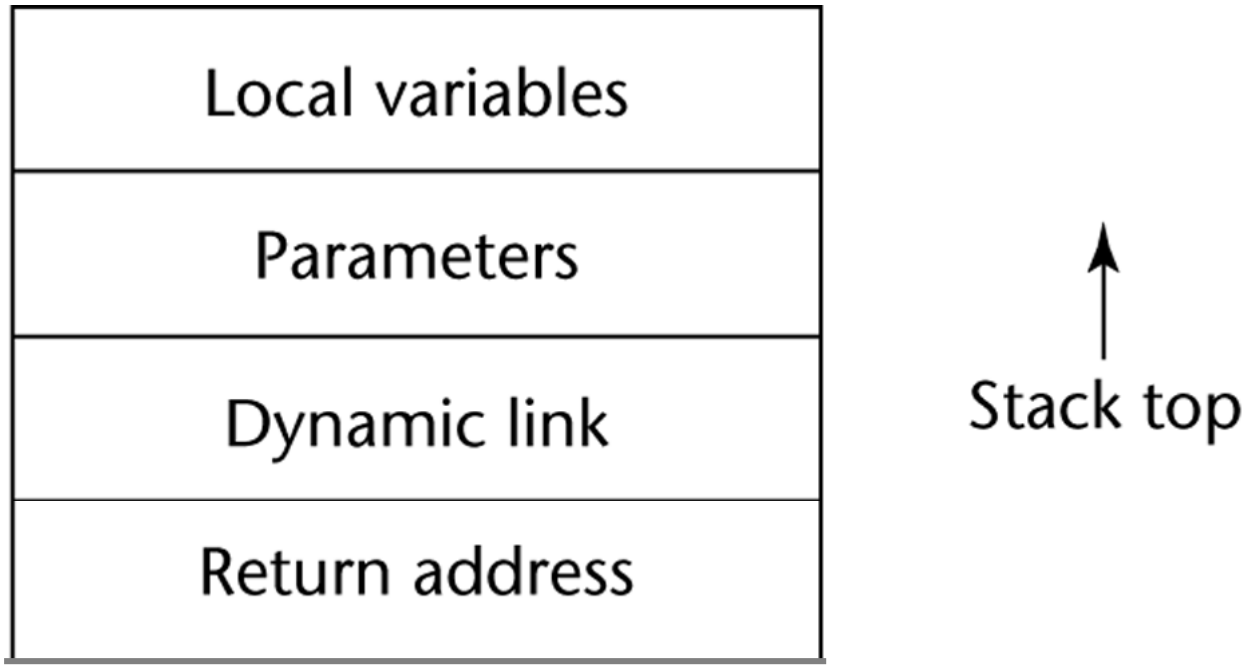
“단순” 부프로그램의 코드와 활성화 레코드



스택-동적 지역변수를 갖는 부프로그램의 구현

- 더 복잡한 활성화 레코드
  - 컴파일러는 지역변수의 묵시적 할당과 해제를 수행하는 코드를 생성해야 함
  - Recursion 지원(부프로그램의 다중 동시 활성화 가능성을 추가시킴)

스택-동적 지역변수를 갖는 전형적인 활성화 레코드

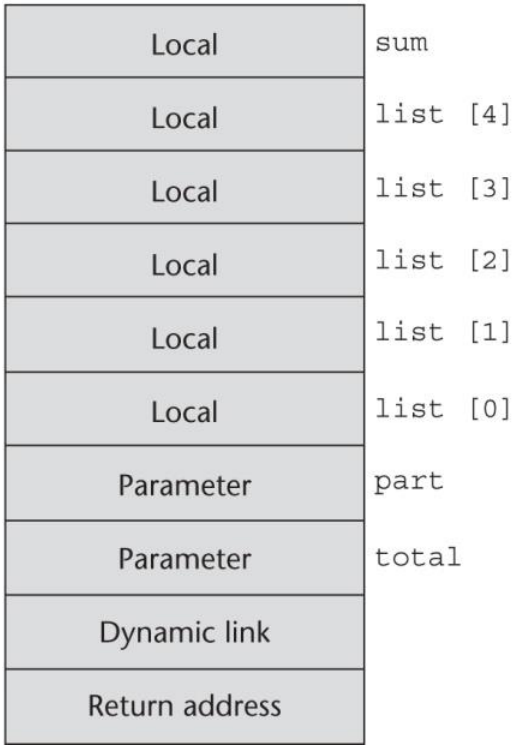


스택-동적 지역변수를 갖는 부프로그램의 구현: 활성화 레코드

- 활성화 레코드 형식은 정적이나, 그 크기는 동적
- *동적 링크(dynamic link)*는 caller의 활성화 레코드 사례의 top을 가리킴
- 활성화 레코드 사례는 부프로그램이 호출될 때 동적으로 생성됨
- 활성화 레코드 사례는 run-time stack에 거주
- *Environment Pointer* (EP)는 run-time system에 의해 유지되며, 현재 실행중인 활성화 레코드 사례의 기준 주소를 갖는다.

An Example: C Function

```
void sub(float total, int part)
{
   int list[5];
   float sum;
   …
}
```



호출/복귀 동작의 의미

- Caller 동작:
  - 활성화 레코드 사례 생성
  - 현재 프로그램의 실행 상태를 저장
  - 매개변수를 계산하고 전달
  - 복귀주소를 called에게 전달
  - 제어를 called에게 이전
- called의 프롤로그 동작:
  - 스택에 있는 old EP를 동적 링크에 저장하고 EP에 새로운 값을 생성
  - 지역변수 할당

호출/복귀 동작의 개정된 의미 (continued)

- called의 에필로그 동작:
  - 값-결과-전달 매개변수나 출력-모드 매개변수가 사용되면 그 매개변수의 값은 대응되는 실매개변수로 이동
  - 부프로그램이 함수라면, 함수값을 caller가 접근가능한 장소로 이동
  - 스택 포인터를 current EP – 1으로 설정하여 스택 포인터를 복원하고 EP를 구 동적 링크로 설정
  - caller의 실행 상태 복원
  - 제어는 caller에게 이전

재귀 없는 예제 프로그램

```
void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}
void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}
void fun3(int q) {
    ...
}
void main() {
    float p;
    ...
    fun1(p);
    ...
}
```

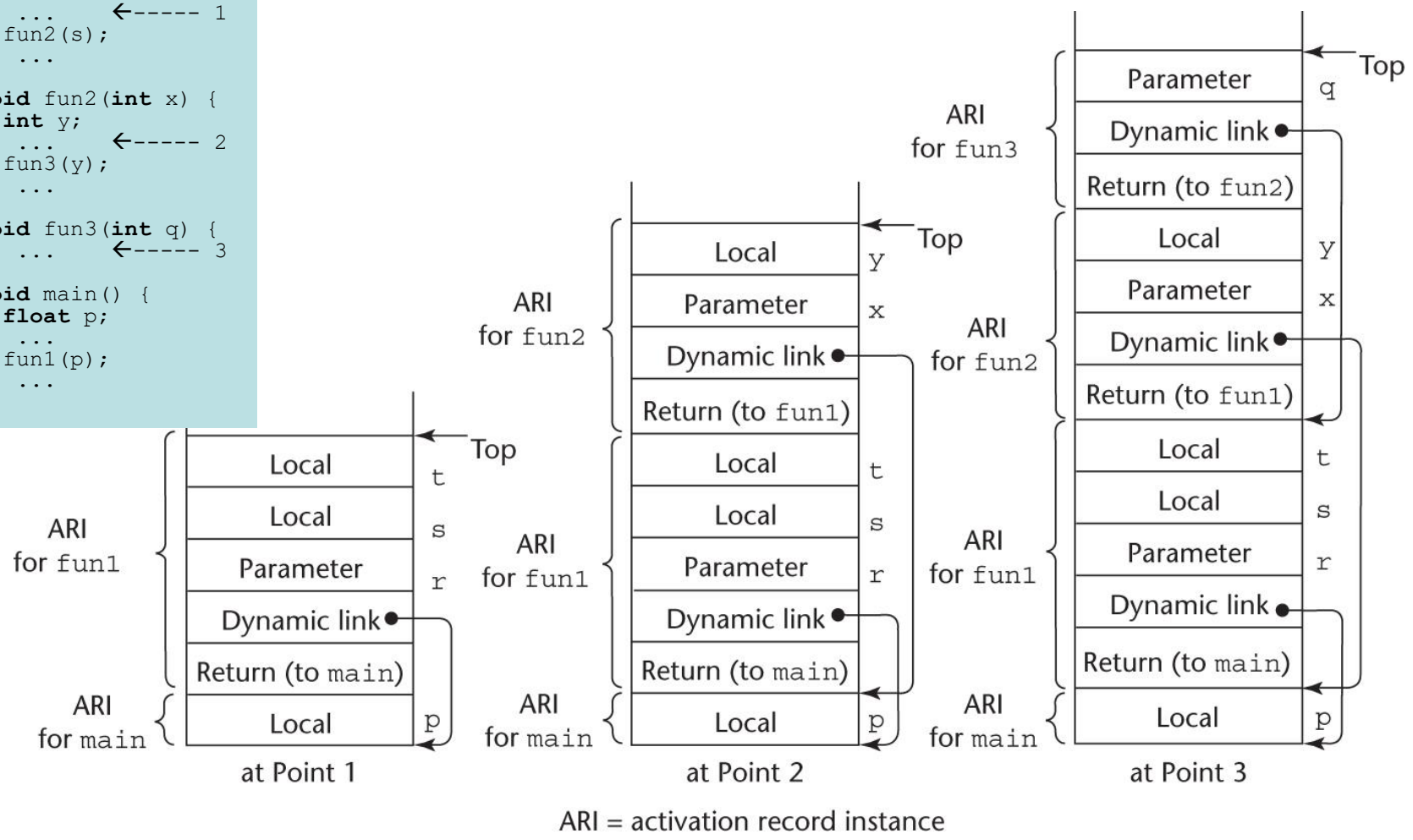
`main` **calls** `fun1`

`fun1` **calls** `fun2`

`fun2` **calls** `fun3`

재귀 없는 예제 프로그램

```
void fun1(float r)
{
  int s, t;
   ...     <----- 1
  fun2(s);
   ...
}
void fun2(int x) {
  int y;
   ...     <----- 2
  fun3(y);
   ...
}
void fun3(int q) {
   ...     <----- 3
}
void main() {
  float p;
   ...
  fun1(p);
   ...
}
```



ARI = activation record instance

동적 체인과 지역 오프셋

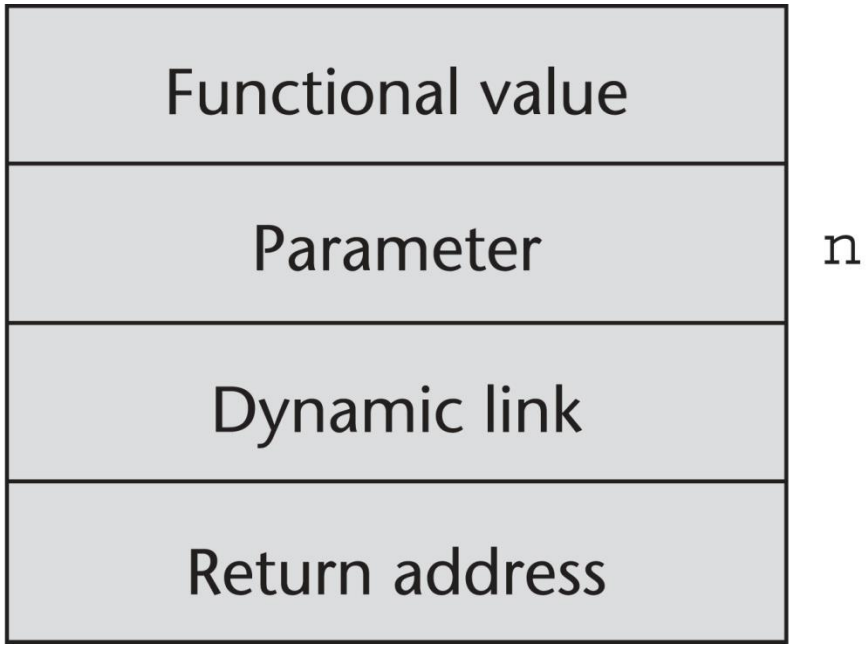
- 주어진 시점에 스택에 있는 동적 링크의 집합을 동적 체인(*dynamic chain*), 또는 호출 체인(*call chain)*
- 지역변수는 활성화 레코드의 시작 지점(주소는 **EP**에 저장)으로부터의 오프셋으로 참조; 이 오프셋을 지역 오프셋(*local_offset)* 이라 함
- 지역변수의 지역 오프셋은 컴파일시간에 컴파일러에 의해 결정

재귀 예제 프로그램

- The activation record used in the previous example supports recursion

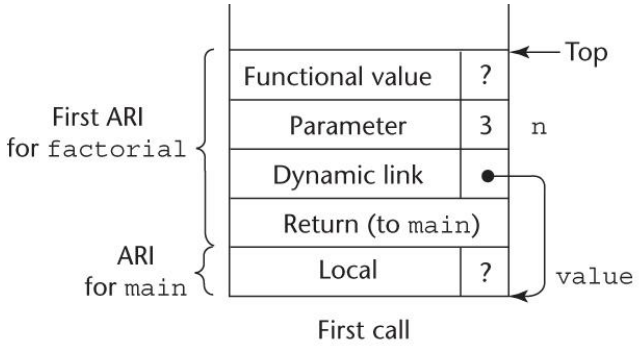
```
int factorial (int n) {
    <------------------------------ 1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
    <------------------------------ 2
}
void main() {
  int value;
  value = factorial(3);
    <------------------------------ 3
}
```

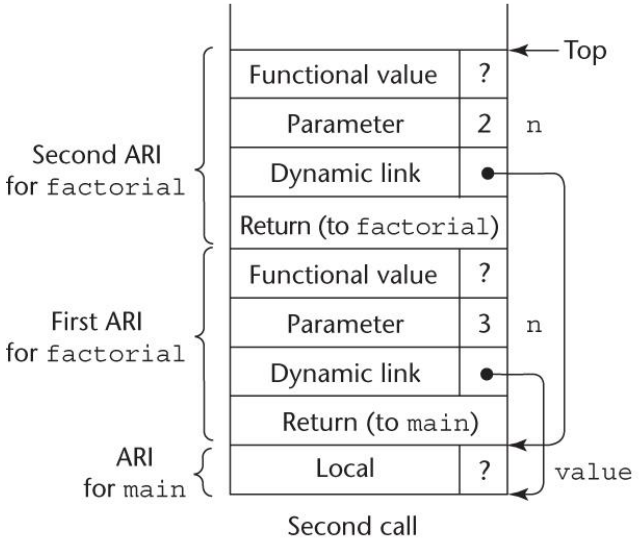
Activation Record for `factorial`

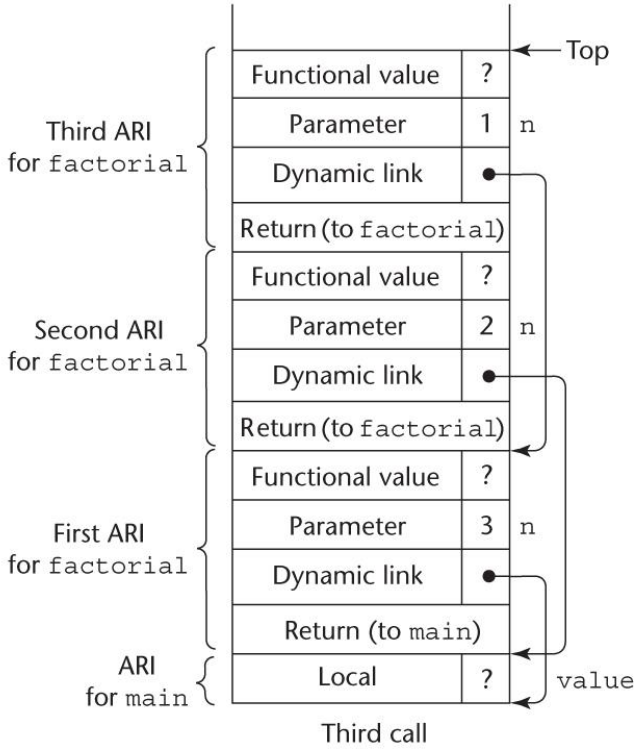


Stacks for calls to `factorial`

```
int factorial (int n) {
                <---------------------------- 1
    if (n <= 1) return 1;
    else return (n * factorial(n - 1));
                <---------------------------- 2
}
void main() {
    int value;
    value = factorial(3);
                <---------------------------- 3
}
```



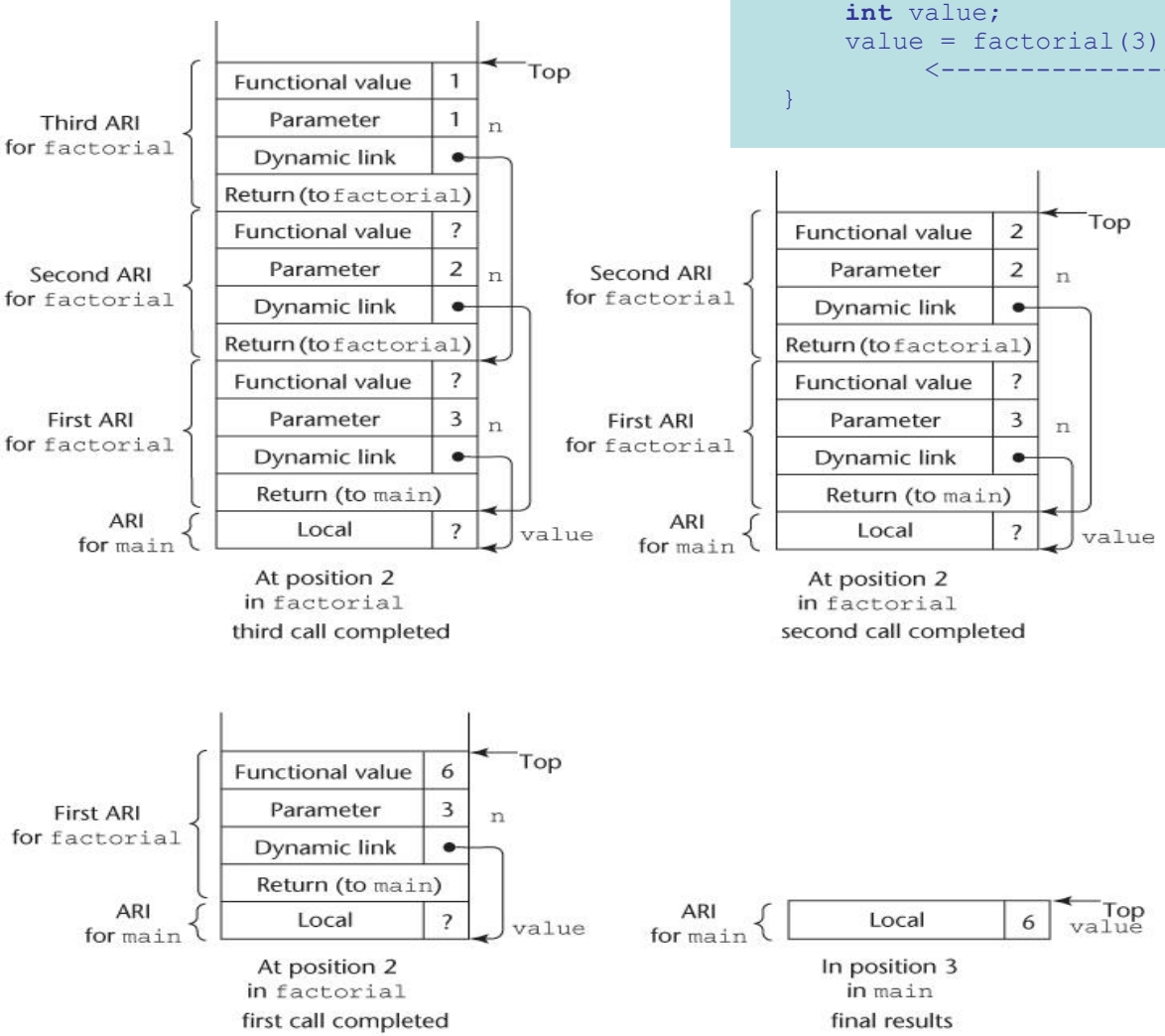




ARI = activation record instance

Stacks for returns from `factorial`

```
int factorial (int n) {
                <---------------------------- 1
     if (n <= 1) return 1;
     else return (n * factorial(n - 1));
                <---------------------------- 2
}
void main() {
     int value;
     value = factorial(3);
                <---------------------------- 3
}
```



ARI = activation record instance

중첩 부프로그램

- Non-C-based static-scoped languages(e.g., Fortran 95+, Ada, Python, JavaScript, Ruby, and Swift)은 스택-동적 지역변수를 사용하며 중첩 부프로그램을 허용
- 비지역적으로 참조될 수 있는 모든 변수는 스택 내 ARI(활성화 레코드 사례)에 거주
- 비지역변수 참조를 위한 위치를 찾는 과정:
  1. 정확한 ARI를 찾음
  2. 그 ARI 내에서 정확한 지역오프셋을 결정

비지역 참조의 위치 찾기

- Finding the offset is easy.
- Finding the correct ARI(activation record instance)
  - 정적 의미규칙은 참조될 수 있는 모든 비지역변수는 스택 내의 어떤 **ARI**에 할당되어 있다는 것을 보장한다

정적 영역

- 정적 체인(*static chain*)은 스택에서의 ARI들을 연결하는 정적 링크의 체인
- 부프로그램 A의 ARI 내에 있는 정적 링크(*static link*)는 A의 정적 조상의 ARI 중 하나를 가리킨다.
- ARI의 정적 체인은 그 정적 조상 모두를 연결한다.
- 정적 깊이(*static_depth*)는 정적 영역이 최외곽 영역으로부터 얼마나 깊게 중첩되었는가를 나타내는 정수

```
def f1():
    def f2():
        def f3():
            …
        # end of f3
        …
    # end of f2
    …
# end of f1
```

정적 영역 (continued)

- 비지역 참조의 체인_오프셋(*chain_offset)* 혹은 중첩_깊이*(nesting_depth)* 는 참조의 정적_깊이와 그 비지역변수가 선언된 영역의 정적_깊이 간의 차이이다.

- 변수의 참조는 정수의 순서쌍 :
  (chain_offset, local_offset) 으로 표시,
  여기서 지역_오프셋은 변수가 참조되는 **ARI** 내에서의 오프셋

Example JavaScript Program

```
function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
        ...
       a = b + c; <----------------------  1
      ...
    }  // end of sub1
    function sub2(x) {
      var b, e;
      function sub3() {
         var c, e;
         …
         sub1();
         ...
         e = b + a; <----------------------  2
      } // end of sub3 ...
      sub3();
      ...
      a = d + e; <-----------------------  3
    } // end of sub2
    ...
    sub2(7);
    ...
  } // end of bigsub
  ...
  bigsub();
  ...
} // end of main
```

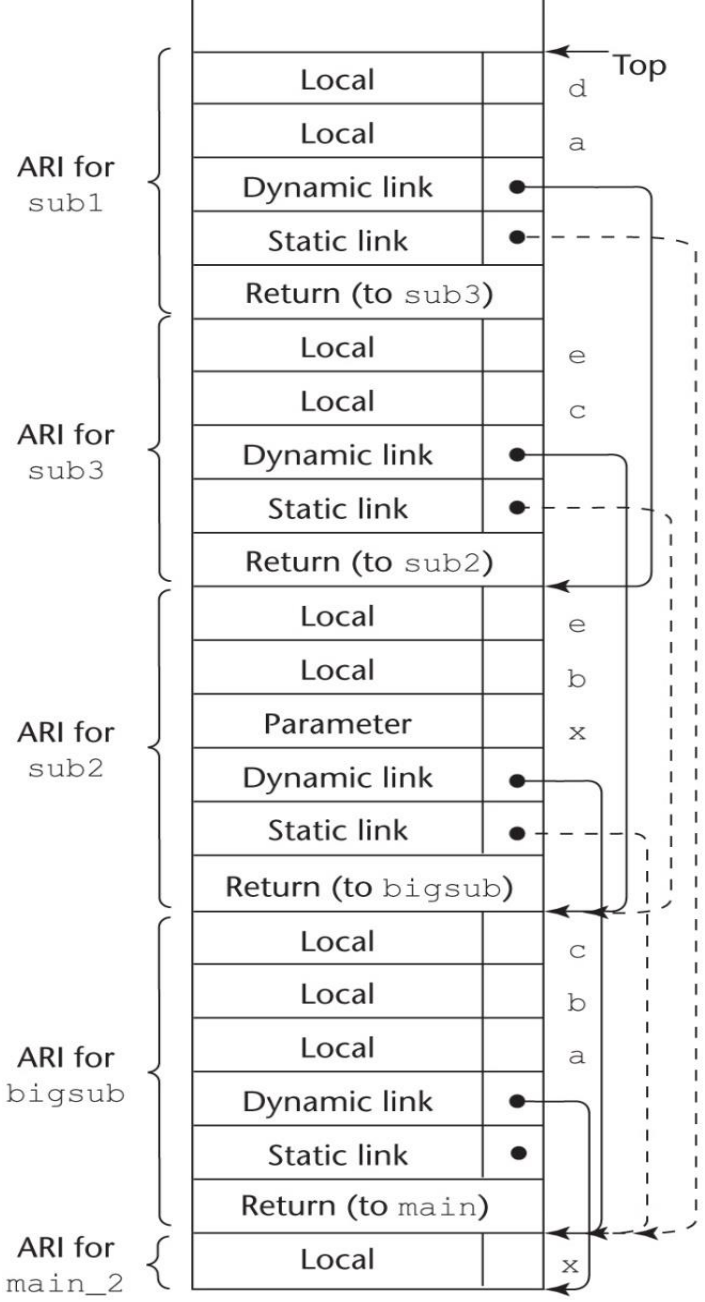
Example JavaScript Program (continued)

- Call sequence for `main`

  `main` calls `bigsub`
  `bigsub` calls `sub2`
  `sub2` calls `sub3`
  `sub3` calls `sub1`

Stack Contents at Position 1

```
function main(){
  var x;
  function bigsub() {
    var a, b, c;
    function sub1 {
      var a, d;
        ...
       a = b + c; <----------------  1
      ...
    } // end of sub1
    function sub2(x) {
      var b, e;
      function sub3() {
        var c, e;
        ...
        sub1();
        ...
        e = b + a; <----------------  2
      } // end of sub3 ...
      sub3();
      ...
      a = d + e; <----------------  3
    } // end of sub2
    ...
    sub2(7);
    ...
  } // end of bigsub
  ...
  bigsub();
  ...
} // end of main
```



ARI = activation record instance

정적 체인 유지관리

- 호출에서,
  - ARI must be built
  - 동적 링크는 old stack top pointer
  - 정적 링크는 정적 조상의 가장 최근의 ARI를 가리켜야 함
    - Two methods:
      1. Search the dynamic chain
      2. Treat subprogram calls and definitions
         like variable references and definitions

정적 체인의 평가

- 문제점:
  1. 중첩깊이가 크다면 비지역 참조는 느리다.
  2. 시간-임계(Time-critical) 프로그램에서 :
     a. 비지역 참조의 비용 추정이 어렵다.
     b. 코드 수정은 중첩 깊이를 변경시킬 수 있고 따라서 비용도 변한다.

블록

- 블록은 변수를 위한 사용자-지정 지역 영역
- An example in C

```
{int temp;
 temp = list [upper];
 list [upper] = list [lower];
 list [lower] = temp
}
```

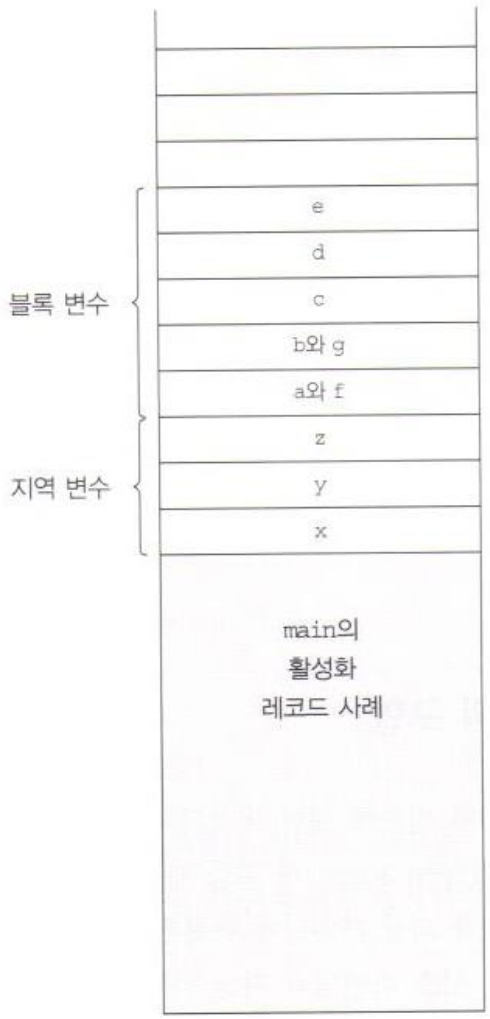
- The lifetime of `temp` in the above example begins when control enters the block
- An advantage of using a local variable like `temp` is that it cannot interfere with any other variable with the same name

블록의 구현

- 두가지 방법:
  1. 블록을 매개변수가 없는 부프로그램(항상 동일한 위치로부터 호출되는) 으로 취급
     - Every block has an activation record; an instance is created every time the block is executed
  2. 블록을 위해 필요한 최대 기억공간이 정적으로 결정될 수 있으므로, 이 공간 량은 활성화 레코드의 지역변수 이후에 할당 가능

블록의 구현 : 지역변수처럼 처리

```
void main() {
  int x, y, z;
  while ( …) {
    int a, b, c;
    …
    while ( …) {
      int d, e;
      …
    }
  }
  while ( …) {
    int f, g;
    …
  }
  …
}
```



동적-영역 규칙의 구현

- 심층 접근*(Deep Access)* : 비지역 참조는 동적 체인상의 **ARI**를 검색함으로 발견
  - 체인의 길이는 정적으로 결정될 수 없음
  - 모든 **ARI**는 변수 이름을 저장해야 함

- 피상 접근*(Shallow Access)* : 중앙 테이블을 사용
  - One stack for each variable name
  - Central table with an entry for each variable name

동적-영역 구현에 심층 접근을 이용한 방법

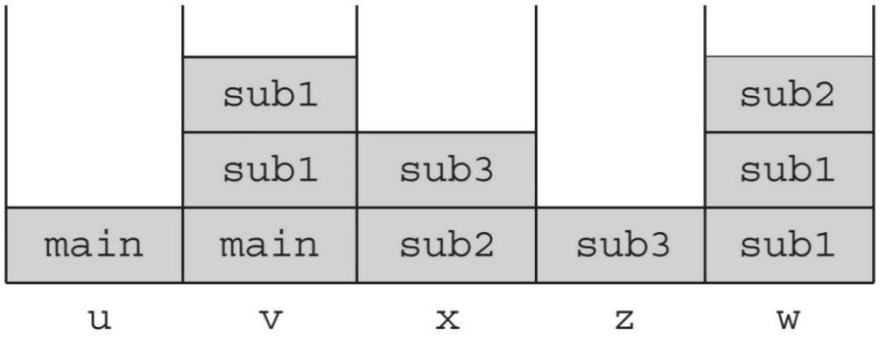
```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```

Call main -> sub1 -> sub1 -> sub2-> sub3

ARI = 활성화 레코드 사례

동적-영역 구현에 피상 접근을 이용한 방법

```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
void sub1() {
  int v, w;
  …
}
void main() {
  int v, u;
  …
}
```



(The names in the stack cells indicate the program units of the variable declaration.)

Call main -> sub1 -> sub1 -> sub2-> sub3

Summary

- Subprogram linkage semantics requires many action by the implementation
- Simple subprograms have relatively basic actions
- Stack-dynamic languages are more complex
- Subprograms with stack-dynamic local variables and nested subprograms have two components
  - actual code
  - activation record

Summary (continued)

- Activation record instances contain formal parameters and local variables among other things
- Static chains are the primary method of implementing accesses to non-local variables in static-scoped languages with nested subprograms
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or thru some central variable table method