

# 제9장 함수 구현

9.1 함수 호출 구현 원리

9.2 인터프리터에서 함수 구현

9.3 컴파일러에서 함수 구현

## 9.1 함수 구현 원리

# 언어 S: 함수 확장

<command> → <decl> | <stmt> | <function>

(형식 매개변수; formal argument)



<function> → fun <type> id(<params>) <stmt> //함수 정의 (선언)

<params> → <type> id {, <type> id}

<type> → int | bool | string | void

<stmt> → ...

| return <expr>; // 반환문

| id(<expr> {, <expr>}); // 함수 호출 (사용) → statement처럼 사용

arguments (실매개변수; actual parameter)

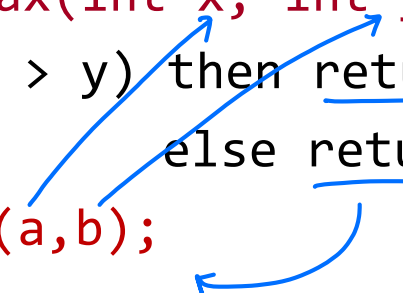
<factor> → ...

| id(<expr> {, <expr>}) // 함수 호출 → factor처럼 사용

# 함수 호출 예

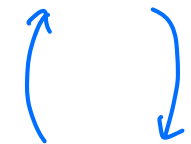
- [예제 1]

```
>> int a = 10;
>> int b = 20;
>> fun int max(int x, int y)
    if (x > y) then return x;
    else return y;
>> print max(a,b);
```



// 함수 정의 (선언)

// 함수 호출 (사용)



- [예제 2]

```
>> fun int fact(int n)
    if (n == 1) then return 1;
    else return n * fact(n-1);

>> print fact(3);
```

# 함수 호출 구현

- 함수 호출 구현을 위해 필요한 사항

- ① 매개변수도 일종의 지역변수
  - 매개변수 메모리 할당
  - 매개변수 전달 < 값
  - 지역 변수 메모리 할당
    - ↪ 반환값을 위한 기억공간 할당 및 저장
- ② 반환 값 및 호출자로의 반환에 필요한 반환 주소 저장
  - 피호출자 함수(callee) 시작 부분으로 제어 이전(goto)
    - A에서 B 호출 : B → callee  
A → caller

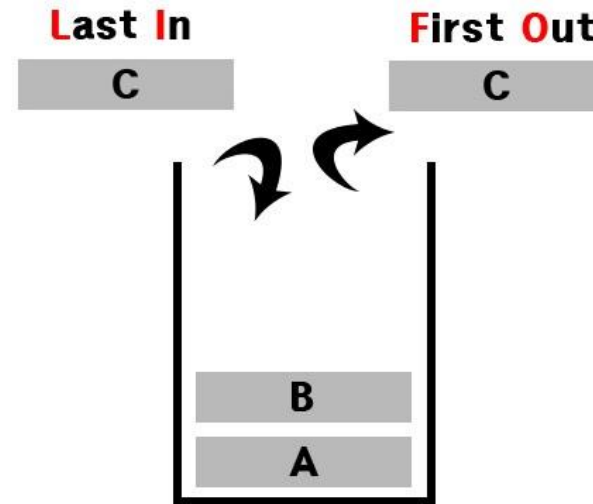
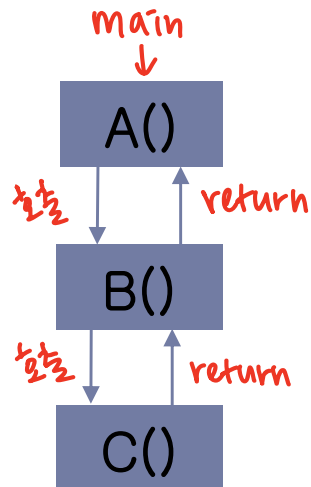
# 함수 반환 구현

---

- 함수 반환 구현에 필요한 사항
  - 지역 변수, 매개변수 등을 위한 기억공간 해제 (free)
  - 반환 값 저장
  - 호출자로의 반환

# 실행시간 스택

- 함수 호출 및 반환

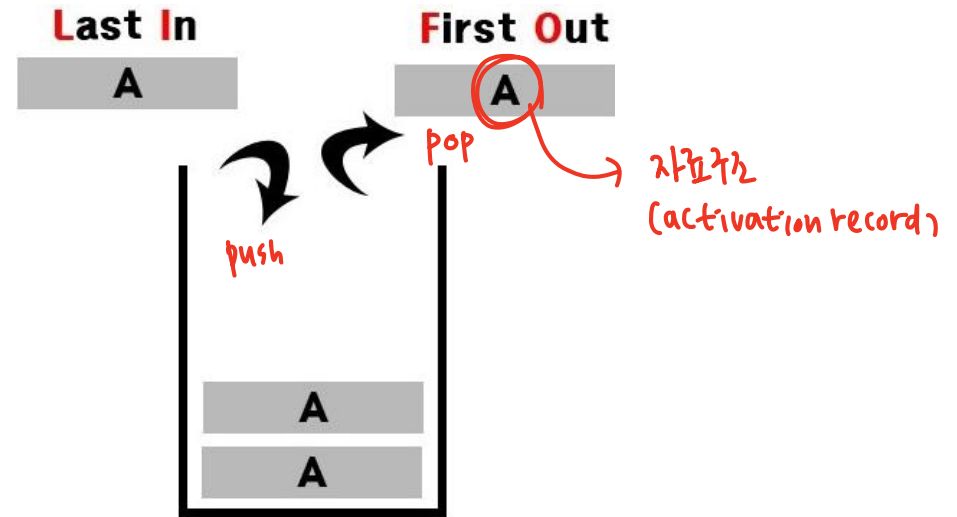
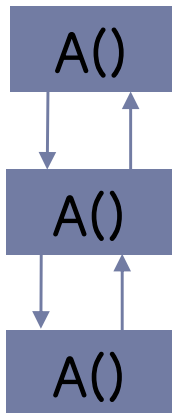


- Last-In-First-Out(LIFO)

- 호출 구현을 위한 자료구조 : 호출할 때마다 만들어야됨
- 실행시간 스택

# 실행시간 스택

- 재귀 함수 호출 및 반환



- Last-In-First-Out(LIFO)
  - 호출 구현을 위한 자료구조
  - 재귀 호출(recursion)



# 실행시간 스택

- 실행시간 스택 Runtime stack

- 함수 호출될 때
  - 새로운 활성 레코드(호출에 필요한 자료구조)가 생성된다.
- 함수가 끝날(반환될) 때
  - 그 활성 레코드를 없앤다.

activation record

- 왜 ?

- 함수의 활성 레코드를 정적으로 할당할 수 없다 (∵ recursion때문에)
- 리커전이 가능함으로 끝나기 전에 다시 호출될 수 있다.
- 새로운 활성 레코드는 함수 호출마다 생성되어야 한다.

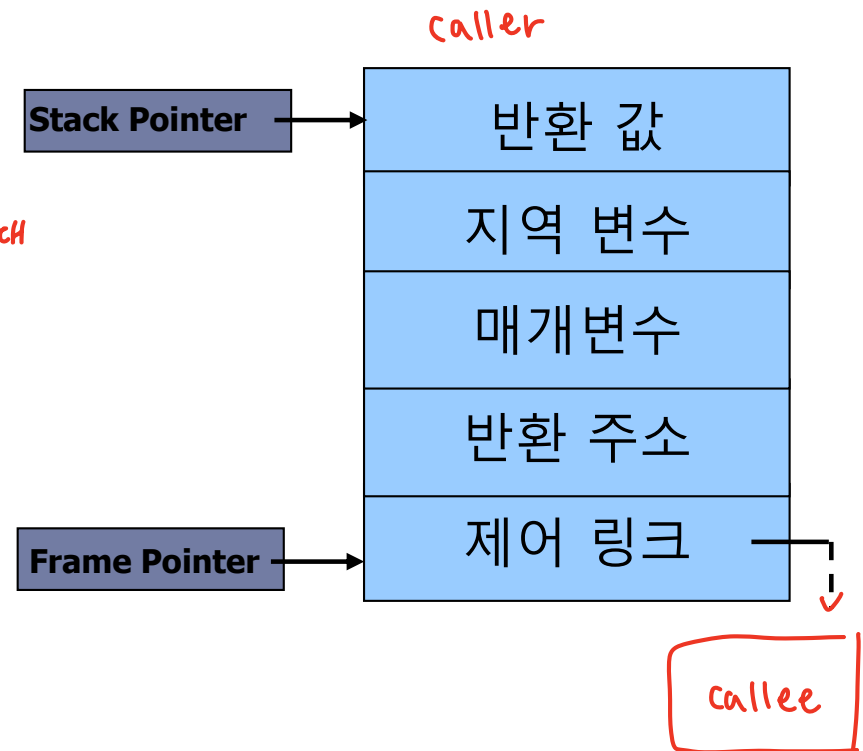
# 활성 레코드 Activation record

## ● 활성 레코드 역할

- 함수 호출/반환에 필요한 정보를 저장하는 자료구조
- 스택 프레임(frame) → 시스템 구현과 관련해 말할 때

## ● 활성 레코드 내용

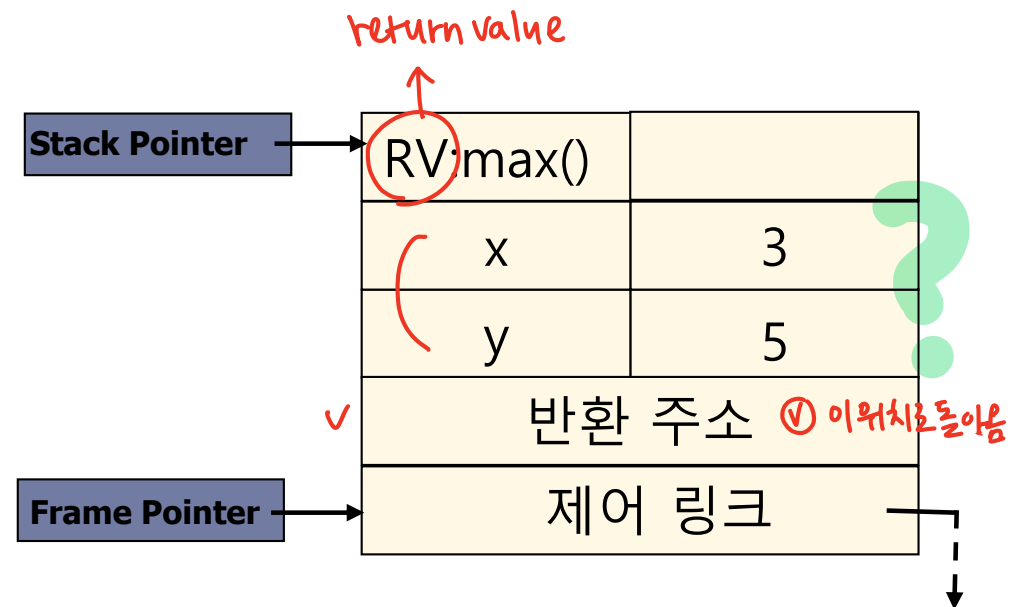
- 매개변수를 위한 기억공간
- 지역변수를 위한 기억공간
- 반환 주소
- 반환 값
- 동적 링크(제어 링크) → 바로 밑의 활성레코드를 가리킴  
(∵ 활성레코드의 크기가 일정하지 않음)
  - 호출자의 활성 레코드



# 함수 호출

```
fun int max(int x, int y)
  if (x > y) then return x;
  else return y;
```

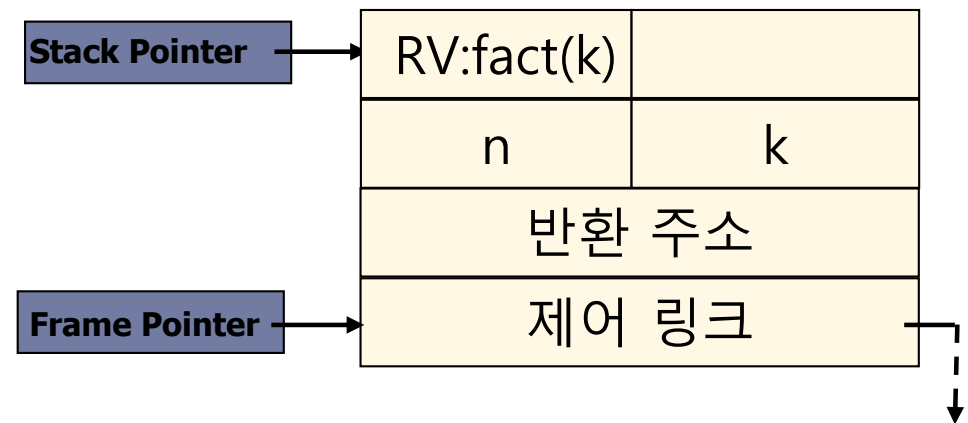
```
print max(3,5);
```



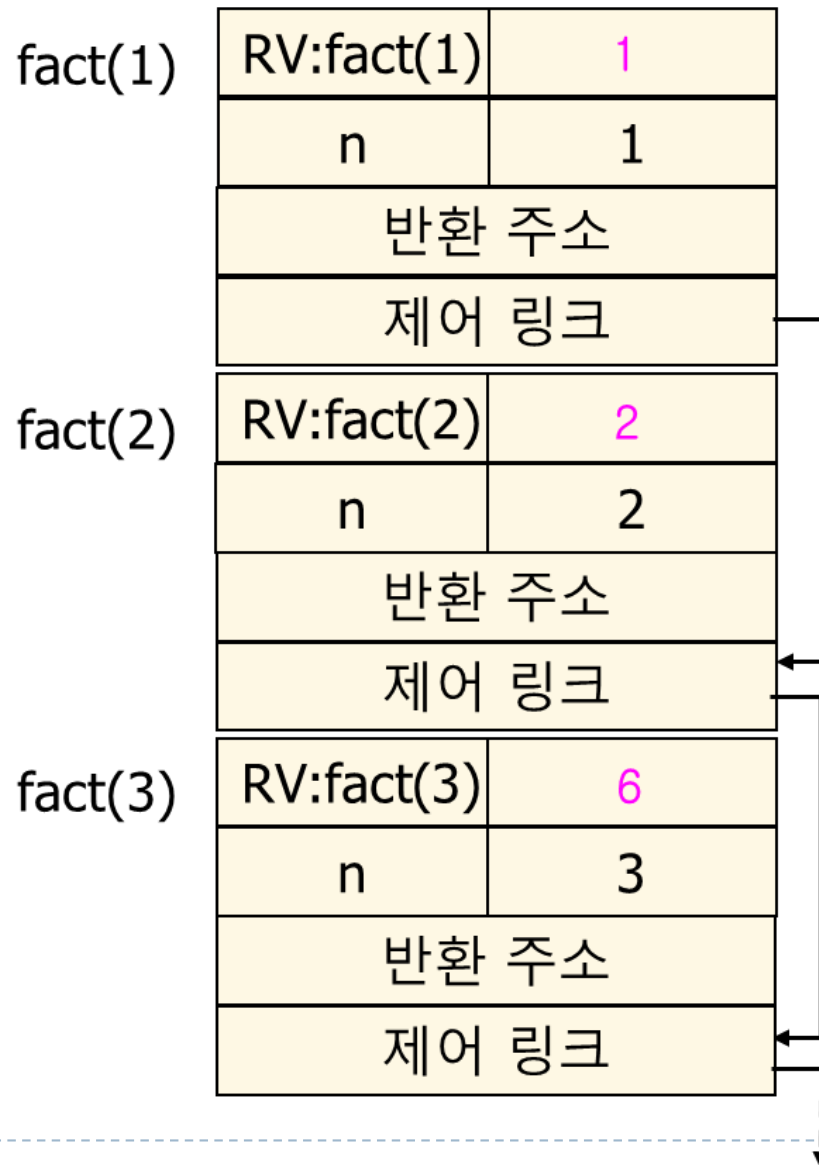
# 재귀 함수 호출

```
fun int fact(int n)
  if (n == 1) then return 1;
  else return n * fact(n-1);

print fact(3);
```



# 재귀 함수 호출/반환



```
fun int fact(int n)
    if (n == 1) then return 1;
    else return n * fact(n-1);
```

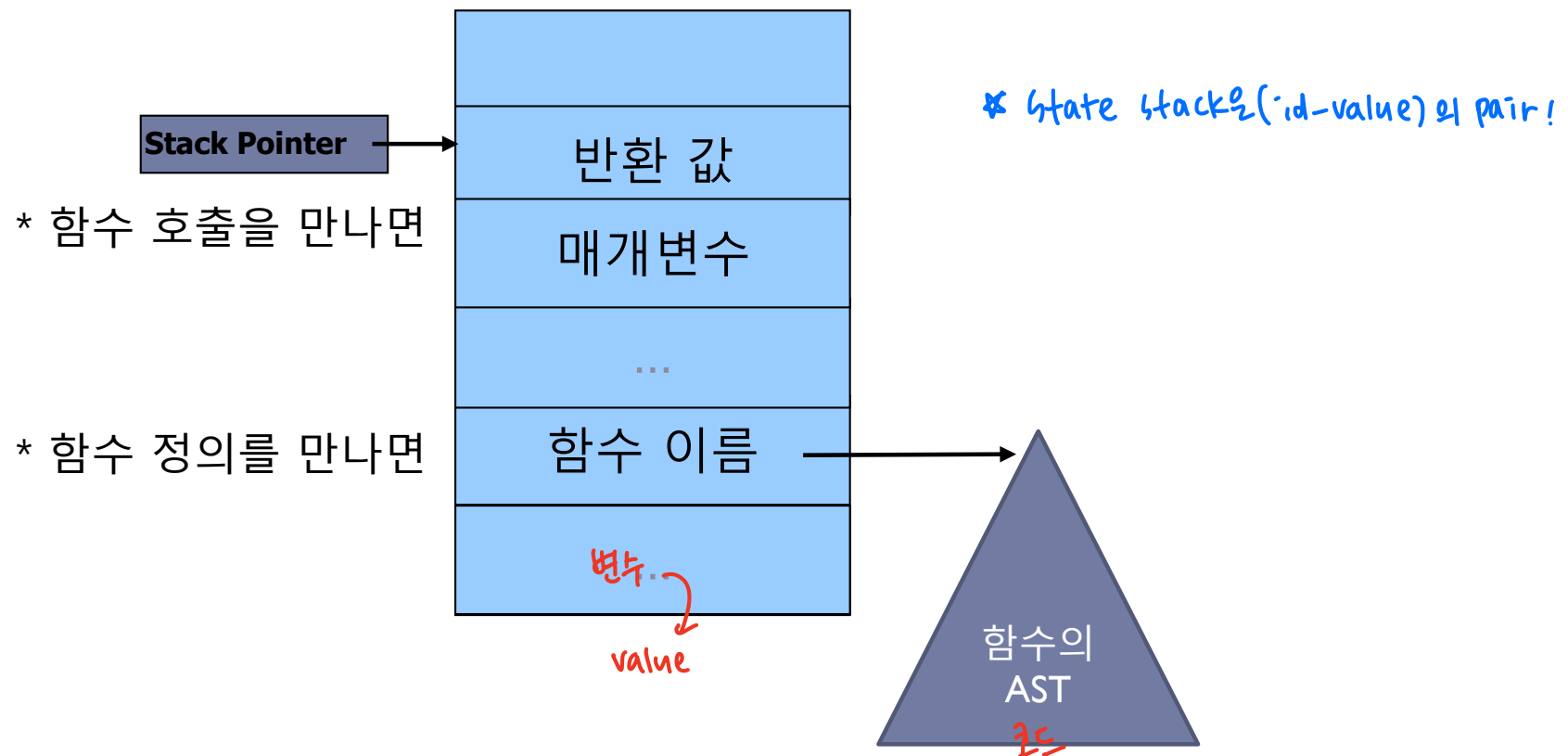
```
print fact(3);
```

## 9.2 인터프리터에서 함수 구현

lang S에서

# 구현 원리

- ① 상태(State) 스택을 실행시간 스택으로 사용
- ② 함수 정의를 만나면 (함수 이름, 함수의 AST)를 PUSH!
- ③ 함수 호출을 만나면 스택 프레임을 구성 → 함수의 코드  
(파싱하고 나면 전부 AST로 바뀌므로)



# 구현 원리

- 함수 정의

- (1) 그 함수의 실행 코드를 기억하기 위해서
- 상태 스택에 함수 이름과 함수의 AST를 push 한다.

↳ 값이면 error  
함수정의 값이 호출한 경우

- 함수 호출

- (1) 스택에서 함수의 실행 코드(AST)를 찾는다.
- (2) 스택에 프레임을 구성하고 매개변수 값을 계산해서 전달한다.
- (3) 호출된 함수의 본체 코드를 실행한다.
- (4) 함수가 반환되면 스택 탑에 반환값이 저장되어 있다.
- (5) 함수 호출이 끝난 후 프레임을 제거한다.

- 함수 반환

- (1) 수식의 값을 계산하고 이 값을 스택 프레임에 저장한다.



# 함수 정의 구현

변수선언, 실행문, 함수정의, ...

- 함수 정의를 만나면 함수 이름과 함수의 AST를 스택에 저장한다.

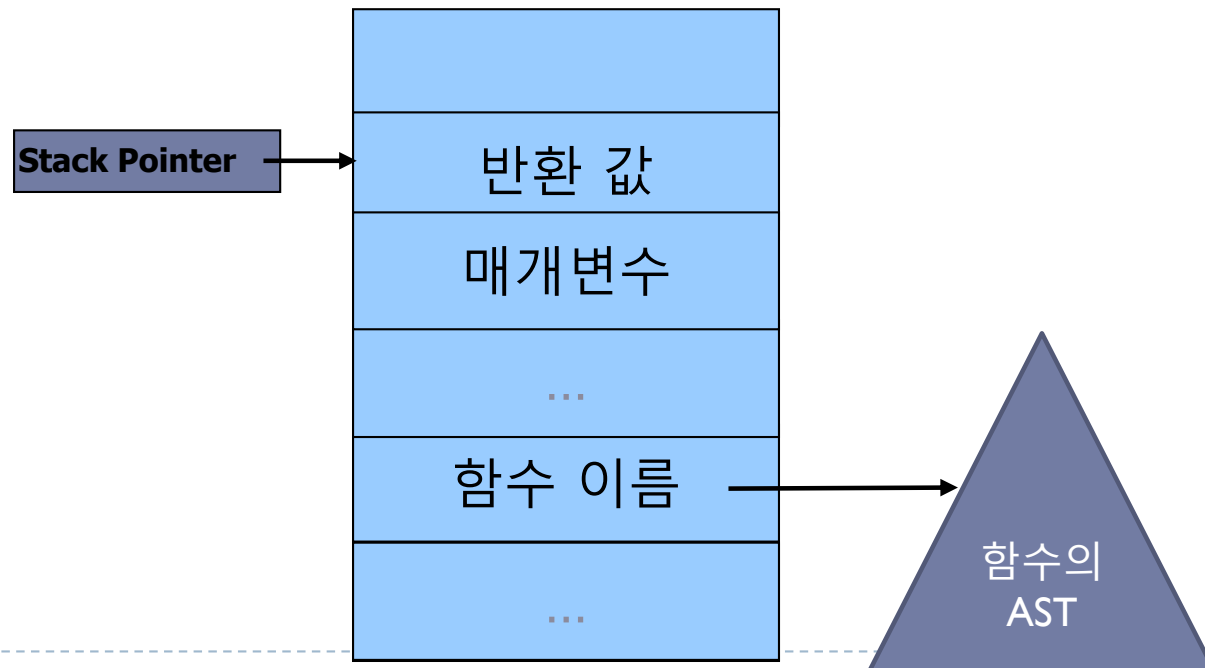
```
State Eval(Command p, State state) {  
  if (p instanceof Function) { // 함수정의일때  
    Function f = (Function) p;  
    state.push(f.id, new Value(f));  
    return state; // 함수이름, 함수의 코드를 나타내는 AST  
  }  
}
```

```
class Value extends Expr { // Value = int value | ... | function value  
  protected boolean undef = true;  
  Object value;  
  Value(Object v) { lab04의 확장  
    ... // 이미 구현된 부분  
    if (v instanceof Function) type = Type.FUN; // 함수의 AST일때  
    value = v; undef = false;  
  }  
}
```

함수의 AST, 값이 정해짐

# 함수 호출 구현

- 함수 호출 `id(<expr> {, <expr>})`
  - (1) 스택에 저장된 함수의 AST를 가져온다.
  - (2) 스택에 프레임을 구성하고 매개변수 값을 전달한다.
  - (3) 해당 함수의 본체를 실행한다.
  - (4) 함수가 반환되면 스택 탑에 반환 값이 저장되며 이를 가져온다.
  - (5) 프레임을 제거한다.



# 함수 호출 구현

함수 호출을 나타내는 AST

```
Value (V) (Call c, State state) { // 함수의 값 계산
    Value v = state.get(c.fid);      // 호출된 함수의 AST 가져오기
    Function f = v.funValue();
    State s = newFrame(state, c, f); // 호출된 함수의 프레임을 스택에 추가
    s = Eval(f.stmt, s);             // 호출된 함수의 본체를 실행
    v = s.peek().val;                // 반환 값 가져오기
    s = deleteFrame(s, c, f);        // 스택에서 프레임 제거
    return v;                        // 반환 값 리턴
}
```

```
State Eval (Call c, State state) {
    // 반환 값이 없는 함수 구현
}
```

# 스택 프레임 구현

newFrame

- 프레임 구성과 매개변수 전달

- (1) 인자 값들을 계산한다.
- (2) 형식 매개변수들을 위한 기억공간 할당한다.
- (3) 인자 값들을 형식 매개변수에 복사한다.
- (4) 프레임에 반환 값 엔트리를 매개변수 바로 위에 추가\*

- 지역 변수 처리?

- 언어 S의 지역 변수는 let 문에 의해서 처리된다.
- 프레임 내의 반환 값 위에 지역 변수 엔트리가 만들어지며(allocate)
- let 문이 끝나면 이들은 없어진다(free).

\* 이 부분은 지역 변수 처리를 반영하기 위해서 교재의 내용을 수정한 것이며  
이를 반영하여 함수 호출 및 반환 관련 구현 내용도 일부 수정함.

# 스택 프레임 구현

- ① State newFrame (State state, Call c, Function f) {  
    Value val[] = new Value[c.args.size()];  
    int i = 0  
    for (Expr e : c.args) // 인자 값을 계산하여 그 값을 val[]에 저장한다.  
        val[i++] = V(e, state);  
    // 현재 상태에 매개변수 기억공간 할당 ✓  
    // 인자의 값을 매개변수에 전달  
    // 프레임에 반환 값을 위한 엔트리 추가  
    // 상태 반환  
}
- ② State deleteFrame (State state, Call c, Function f) {  
    // 프레임에서 반환 값 엔트리 제거  
    // 프레임에서 매개변수를 위한 기억공간 제거(free 사용)  
    // 상태 반환

# 함수 반환 구현

- 함수 반환 `return <expr>`

- 수식의 값을 계산하고 그 값을 프레임에 반환 값으로 저장한다.

```
State Eval (Return r, State state) {  
    // 수식의 값을 계산하고 그 값을 프레임에 반환 값으로 저장  
    // 상태 반환  
}
```

- \* 프레임 내에 반환 값의 위치는 매개변수 바로 위이며
- \* 반환 값 위에 지역 변수를 위한 엔트리가 있을 수 있다.

# 유효범위 규칙 구현

- [예제 4]

```
1 int x = 0;  
2 fun void g(int y)  
3   x = x + y * y;  
4 fun void f(int z)  
5   let int x = 10; in  
6   g(z);  
7 end;  
8 f(5);
```

- 유효범위 규칙?

함수 g의 프레임	y	5
함수 f의 프레임	x	10
	z	5
전역 변수 영역	x	0

그림 9.7 함수 g가 호출됐을 때 상태 스택

동작유효범위규칙 구현이정적보다쉬움

# 정적 유효범위 규칙 구현 (참고)

---

- (1) 접근할 변수를 찾을 때는 먼저 스택 탑에 있는 스택 프레임에서 찾는다. 여기서 찾으면 이는 지역변수이다.
- (2) 여기서 찾지 못하면 이는 지역 변수가 아니고 전역 변수이므로 전역 변수 영역에서 찾아야 한다.
- (3) 이를 위해서는 상태 스택 내에 지역 변수가 저장되는 스택 프레임과 전역 변수가 저장되는 전역 변수 영역을 구분할 수 있어야 한다



## 9.3 컴파일러에서 함수 구현

# 함수 구현 방법

---

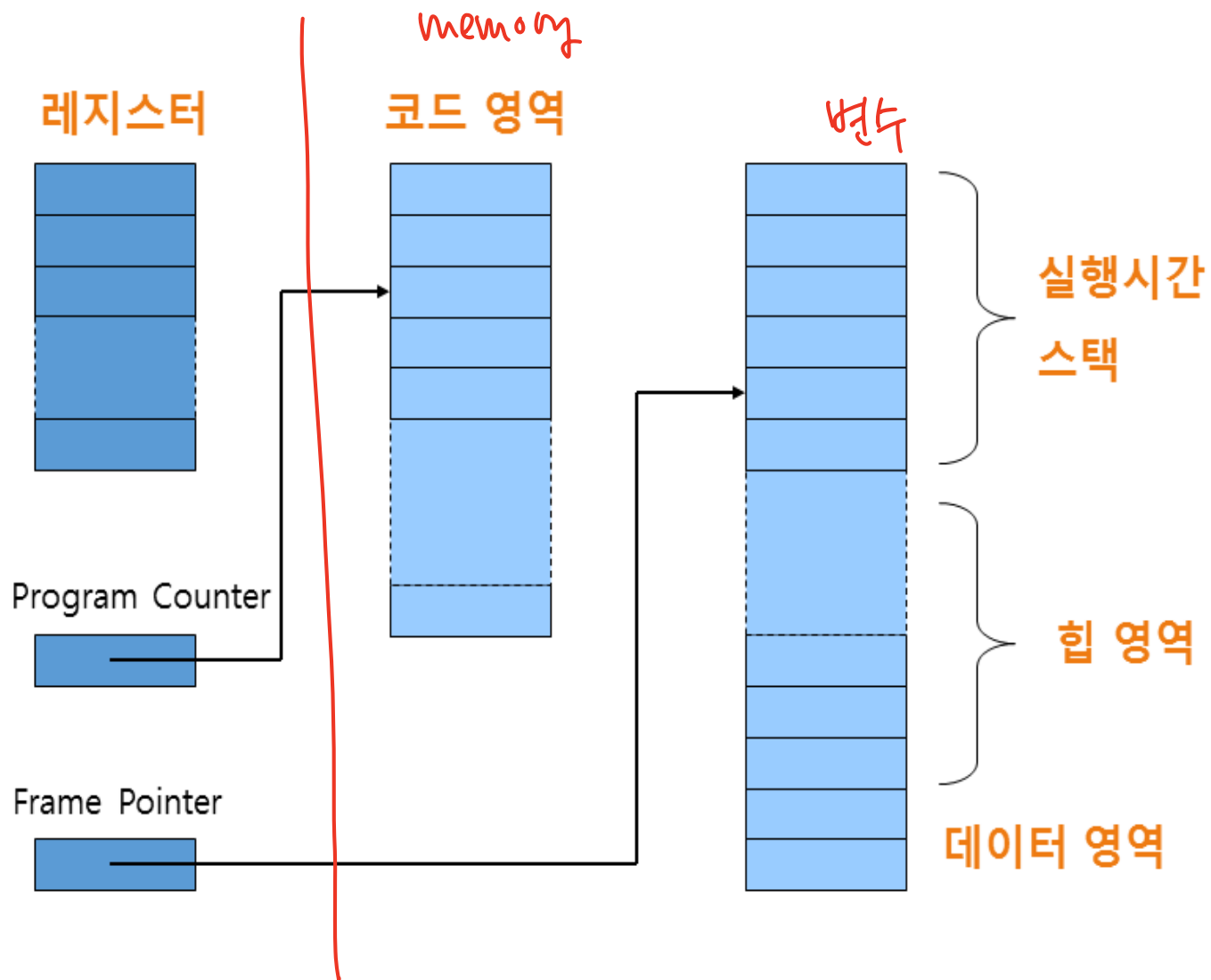
- 컴파일러 사용

- 프로그램 실행을 위한 Hardware Machine Model
  - 레지스터, 코드, 스택, 힙
- 컴파일 후 기계어 코드가 실행된다.
  - 코드 생성
  - 변수를 위한 기억공간 할당
    - 지역변수, 매개변수, 비지역변수, 동적변수, ...
  - 함수 호출 구현을 위한 코드 생성

- 인터프리터 사용

- 인터프리터 내에서 해석하여 실행한다.
- 인터프리터 내에서 함수 호출도 구현한다.

# 컴퓨터 메모리 구조



# 메모리 영역

---

- 레지스터
  - Program Counter(PC): 코드 영역에 대한 포인터
  - Frame pointer: 스택에 대한 포인터
- 코드(텍스트) 영역 Code(Text) segment
  - 프로그램을 구성하는 기계어 코드를 저장
- 실행시간 스택
  - 주로 함수 호출을 구현하기 위해서 사용되는 기억공간
  - 지역변수, 매개변수, 반환값, 반환주소 등을 위한 기억공간

# 메모리 영역

---

- 힙 영역 Heap

- 동적 메모리 할당을 위한 기억공간으로 사용된다.
- malloc() in C, new() in Pascal, Java

- 데이터 영역 Data segment

- 정적 변수, 전역 변수
- `int maxcount = 99; // 초기화된 변수(initialized)`
- `long sum[1000]; // 초기화되지 않은 변수(uninitialized)`

# 어떤 실행 환경이 필요할까?

---

- 실행시간 스택 Runtime stack

- 함수 호출될 때

- 새로운 활성 레코드(호출에 필요한 정보 포함)가 생성된다.

- 함수가 끝날(반환될) 때

- 그 활성 레코드를 없앤다.

- 왜 ?

- 함수의 활성 레코드를 정적으로 할당할 수 없다

- 리커전이 가능함으로 끝나기 전에 다시 호출될 수 있다.

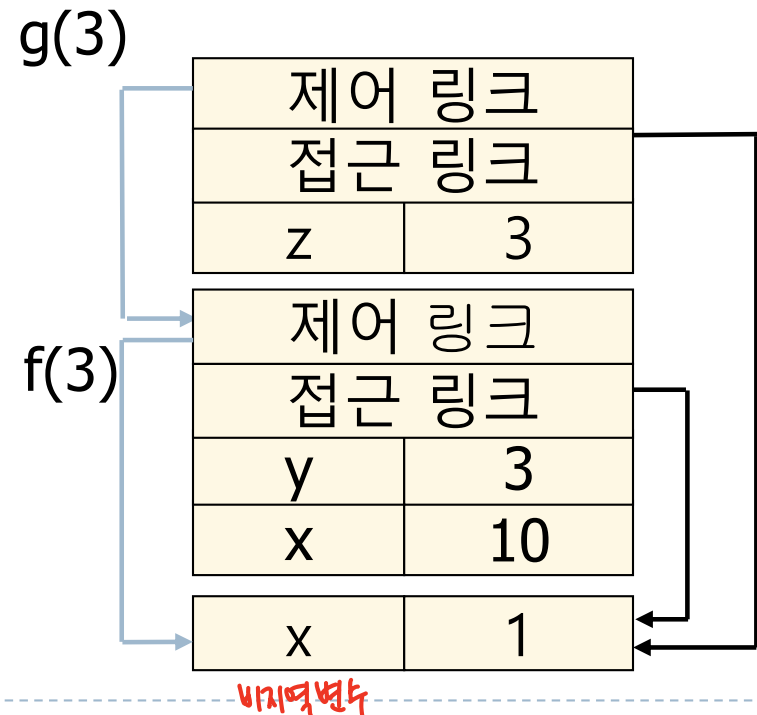
- 새로운 활성 레코드는 함수 호출마다 생성되어야 한다.

# 비지역 변수와 정적 유효범위 규칙

<stmt> → ...  
| let <decls>  
| <funcs> // 블록 내에 함수 정의  
in <stmts>  
end;  
<funcs> → {<function>}

→ 1과 내에서 함수가 있을 수 있음

```
let int x=1;  
  fun void g(int z)  
    return x+z;  
  fun void f(int y)  
    let int x = 10; in  
    return g(y)  
  end  
in  
  f(3);
```



# 비지역 변수와 정적 링크

- 제어 링크Control link
  - 호출 관계를 나타내는 동적 링크Dynamic link
  - 호출자의 활성레코드(바로 전 활성레코드)에 대한 포인터
- 접근 링크Access link
  - 정적 유효범위 구현을 위한 정적 링크Static link
  - 비지역변수 접근을 위한 포인터
  - 프로그램 내의 함수 정의의 포함 관계
  - 호출된 함수가 정의된 함수의 활성 레코드에 대한 포인터



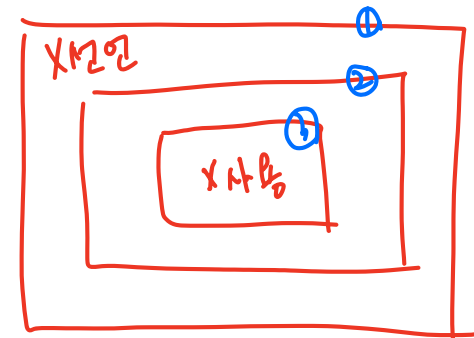


# 지역 변수 접근

- 컴파일러에서 지역 변수 접근
  - 지역 변수가 사용된다는 것은
  - 이를 선언한 함수가 현재 호출 되어 있음을 의미한다.
  - 따라서 fp가 해당 함수의 활성 레코드를 가리키고 있다.
- 지역 변수의 주소
  - $(fp) + offset$
  - 프레임 포인터가 가리키는 주소 + 상대위치

# 비지역 변수 접근

- 컴파일러에서 비지역변수 접근
  - 정적 링크(접근 링크)를 사용한다.
  - 호출된 함수의 바로 바깥 함수 즉
  - 호출된 함수를 정의한 함수를 가리킨다.
- 접근 체인 access chaining
  - 비지역 변수를 찾기 위해 접근 링크를 따라간다.
- 비지역 변수  $x$ 의 주소  $\text{addr}(x)$ 
  - 접근 체인에 의해 도달한 활성 레코드 주소 + 변수  $x$ 의 상대 위치
- 몇 번 접근 링크를 따라 가야 하는가 ?
  - 변수  $x$ 를 사용하는 함수의 중첩 레벨 - 변수  $x$ 를 선언한 함수의 중첩 레벨



⇒  $h-1=2$   
접근링크는 2번 따라간다.

# 실습문제 #5

---

1. 다음의 문법을 기준으로 하여 인터프리터에 함수 관련 기능들을 구현하시오.

<command> → ... | <function>                      // 함수 정의

<function> → fun <type> id(<params>) <stmt>

<params> → <type> id {, <type> id}

<type> → int | bool | string | void

<stmt> → ...

    | id(<expr> {, <expr>});                      // 함수 호출

    | return <expr>;                              // 리턴문

<factor> → ... | id(<expr> {, <expr>})              // 함수 호출