



# Functions

- Functions are parameterized pieces of code with a single entry point.
- The function body is usually executed in the context of its own local scope.
- When the function exists that local scope disappears
- Function declaration and calling works analogously to variable declaration and usage.
- Function arguments that appear in the *declaration* are called “formal parameters”
- Function arguments that appear in the function *call* are called “actual parameters”.

# Parameters



Terminology: Example: Java, C, C++

Function Declaration { int plus (int a, int b) ← Formal Parameters  
{ return a + b; } Function Body  
}

Function Call { int x = plus(1,2); ← Actual Parameters  
...  
}

Observation: in function declaration formal parameters act as placeholders for the values of actual parameters.



# Two Fundamental Questions

- How is the correspondence between actual and formal parameters established?
- How is the value of an actual parameter transmitted to a formal parameter?

# Correspondence



Most programming languages use positional parameters; the first actual parameter is assigned to the first formal parameter, the second actual parameter is assigned to the second formal parameters, *etc.*

```
      1 2
int x = plus(1,2);
           1 2
           ↓ ↓
int plus (int a, int b)
{
    return a + b;
}
```

# Correspondence

Some languages such as Ada provide keyword parameters.

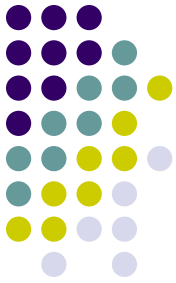
Example: Ada

```
FUNCTION Divide(Dividend:Float, Divisor:Float) RETURN Float IS
BEGIN
  RETURN Dividend/Divisor;
END
```

```
...
Foo = Divide(Divisor => 2.0, Dividend => 4.0);
...
```

2nd formal  
parameter  
becomes 2.0

1st formal  
Parameter  
Becomes 4.0



# Parameter Value Transmission



- Two of the most popular techniques
  - By value
  - By reference



# I. By Value

For by-value parameter passing, the formal parameter is just like a local variable in the activation record of the called method, with one important difference: it is initialized using the value of the corresponding actual parameter, before the called method begins executing.

- Also called ‘copy-in’
- Simplest method
- Widely used
- The only method in Java



## II. By Reference

For passing parameters by reference, the memory address (reference) of the actual parameter is computed before the called method executes. Inside the called method, that memory address (reference) is used as the memory address (reference) of the corresponding formal parameter. In effect, the formal parameter is an alias for the actual parameter — another name for the same memory location.

- One of the earliest methods: Fortran
- Most efficient for large objects
- Still frequently used; C++ allows you define calls by reference





# Functions

- We extend our Cuppa2 language (our language with variable declarations and scoping) to include function declaration and calling, and call it Cuppa3:
  - Declaration: declare `inc(x)` return `x+1`;
  - Call Statement: `inc(3)`
  - Call as expression: `4 + inc(3)`
- We implement positional parameter correspondence and call-by-value.



# Functions

Listing 8.1: A grammar for the Cuppa3 language.

```
1  stmt_list : (stmt)*
2
3  stmt : declare ID \( formal_args? \) stmt ←
4        | declare ID (= exp)? ;?
5        | ID \( actual_args? \) ;? ←
6        | ID = exp ;?
7        | get ID ;?
8        | put exp ;?
9        | return exp? ;?
10       | while \( exp \) stmt
11       | if \( exp \) stmt (else stmt)?
12       | \{ stmt_list \}
13
14  exp : exp_low
15  exp_low : exp_med ((= | =<) exp_med)*
16  exp_med : exp_high ((\+ | -) exp_high)*
17  exp_high : primary ((\* | /) primary)*
18
19  primary : INTEGER
20          | ID
21          | ID \( actual_args? \) ←
22          | \( exp \)
23          | - primary
24          | not primary
25
26  formal_args : ID (, ID)*
27  actual_args : exp (, exp)*
28
29  ID : <any valid variable name>
30  INTEGER : <any valid integer number>
```

# Example Programs



```
declare add(a,b)
{
    return a+b;
}

declare x = add(3,2);
put x;
```

```
declare seqsum(n)
{

    declare add(a,b)
    {
        return a+b;
    }

    declare i = 1;
    declare sum = 0;

    while (i <= n)
    {
        sum = add(sum,i);
        i = i + 1;
    }

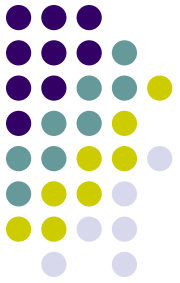
    put sum;
}

seqsum(10);
```

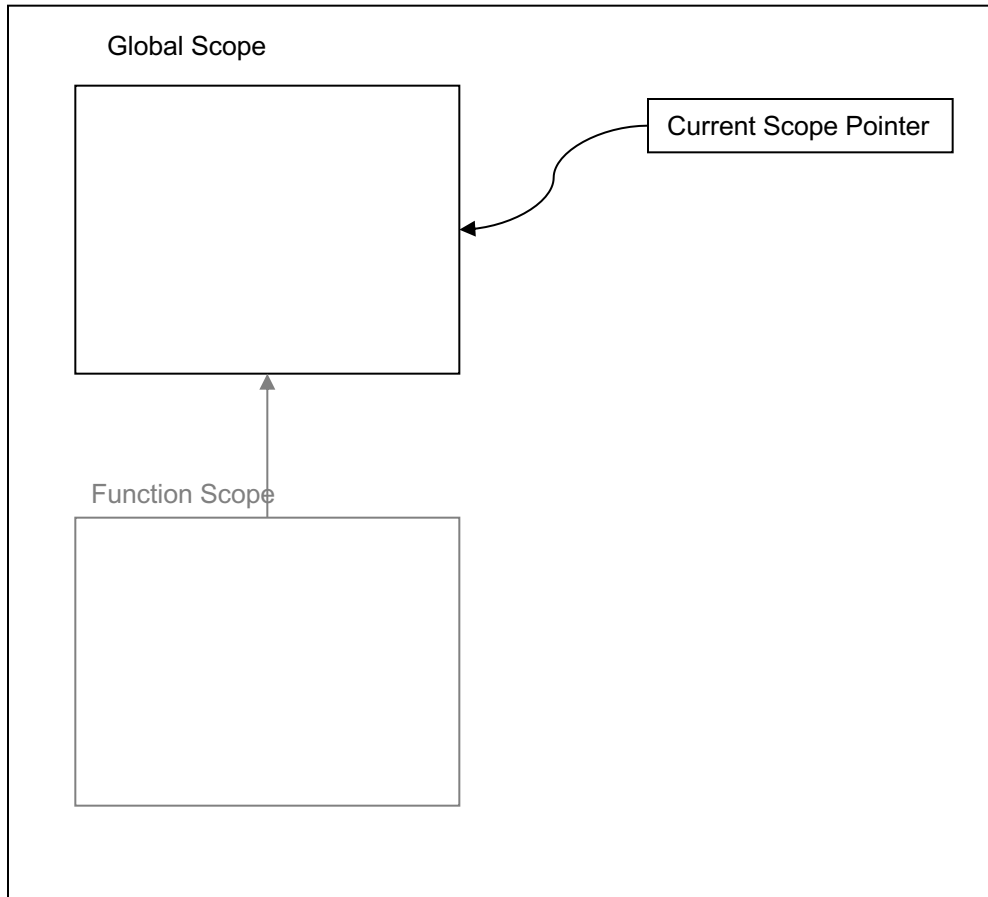
```
// recursive implementation of factorial
declare fact(x)
{
    declare y;
    if (x <= 1)
        return 1;
    else
    {
        y = x*fact(x-1);
        return y;
    }
}

declare v;
get v;
put fact(v);
```

# Programs with Functions



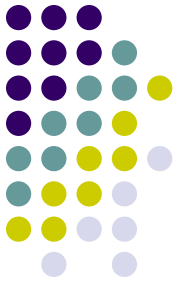
Symbol Table



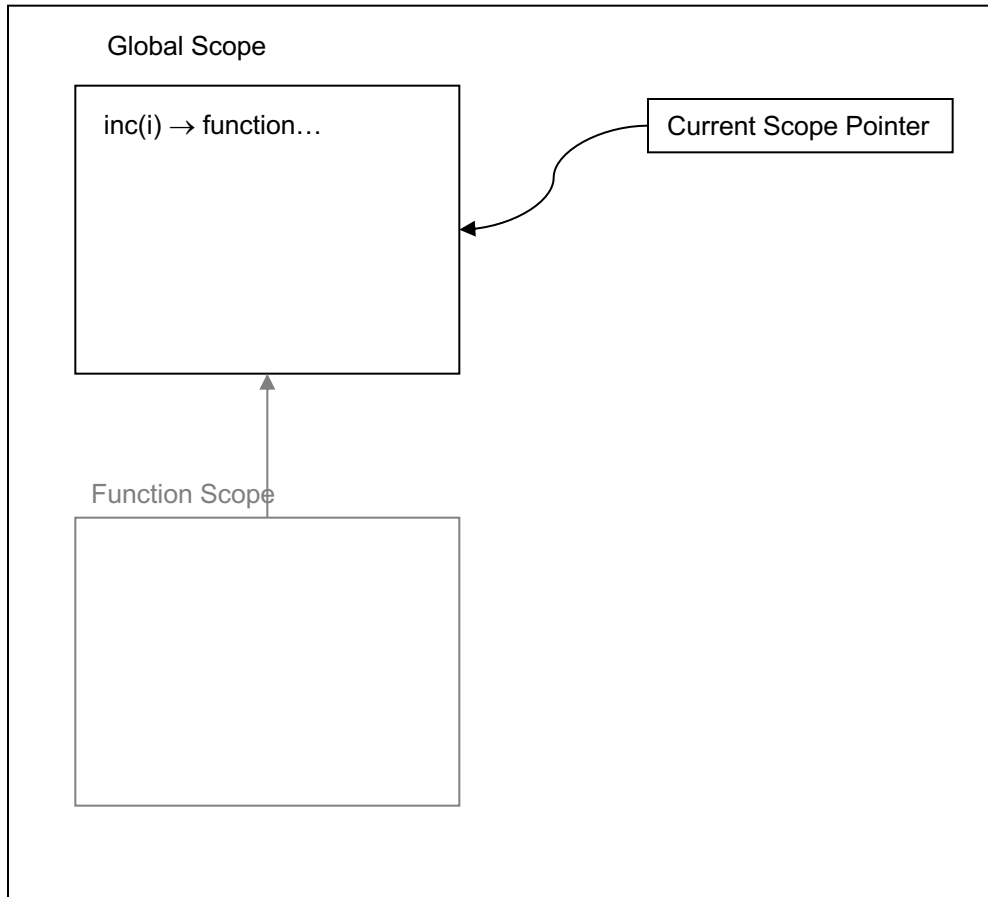
```
declare inc(i) return i+1;
```

```
declare x = 10;  
declare y;  
y = inc(x);  
put y;
```

# Programs with Functions

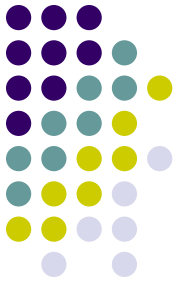


Symbol Table

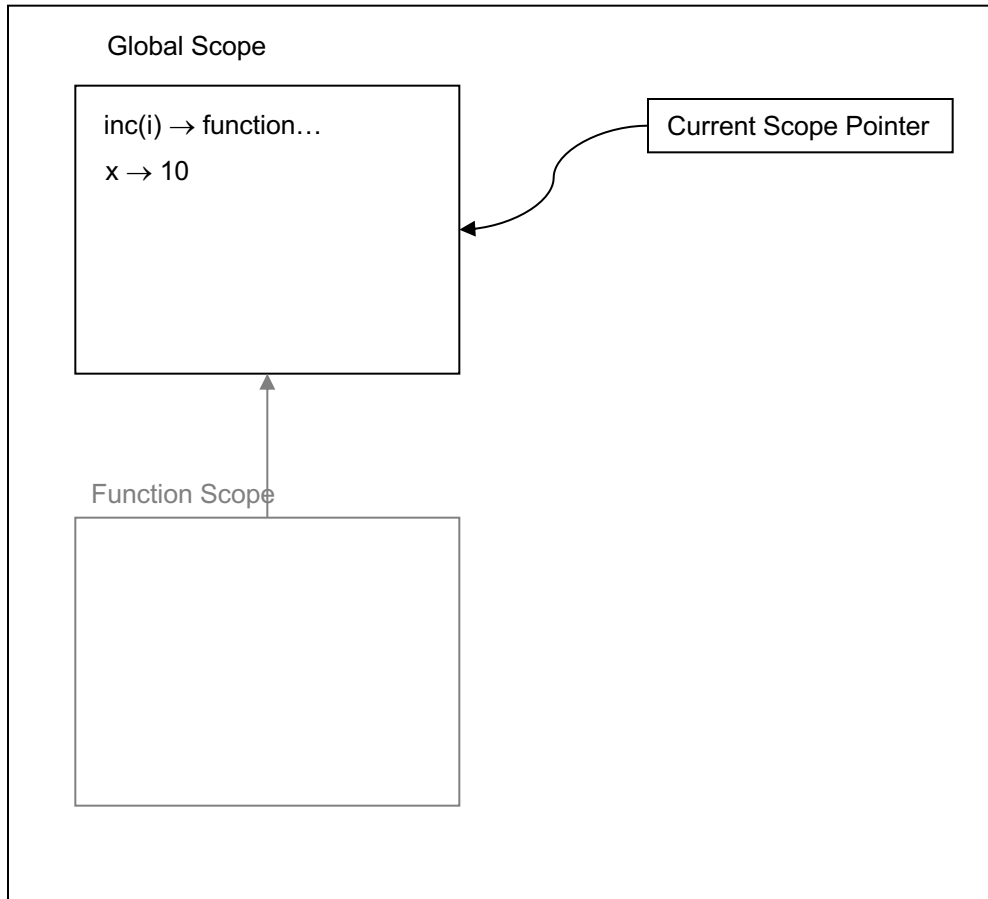


→ declare inc(i) return i+1;  
  
declare x = 10;  
declare y;  
y = inc(x);  
put y;

# Programs with Functions



Symbol Table

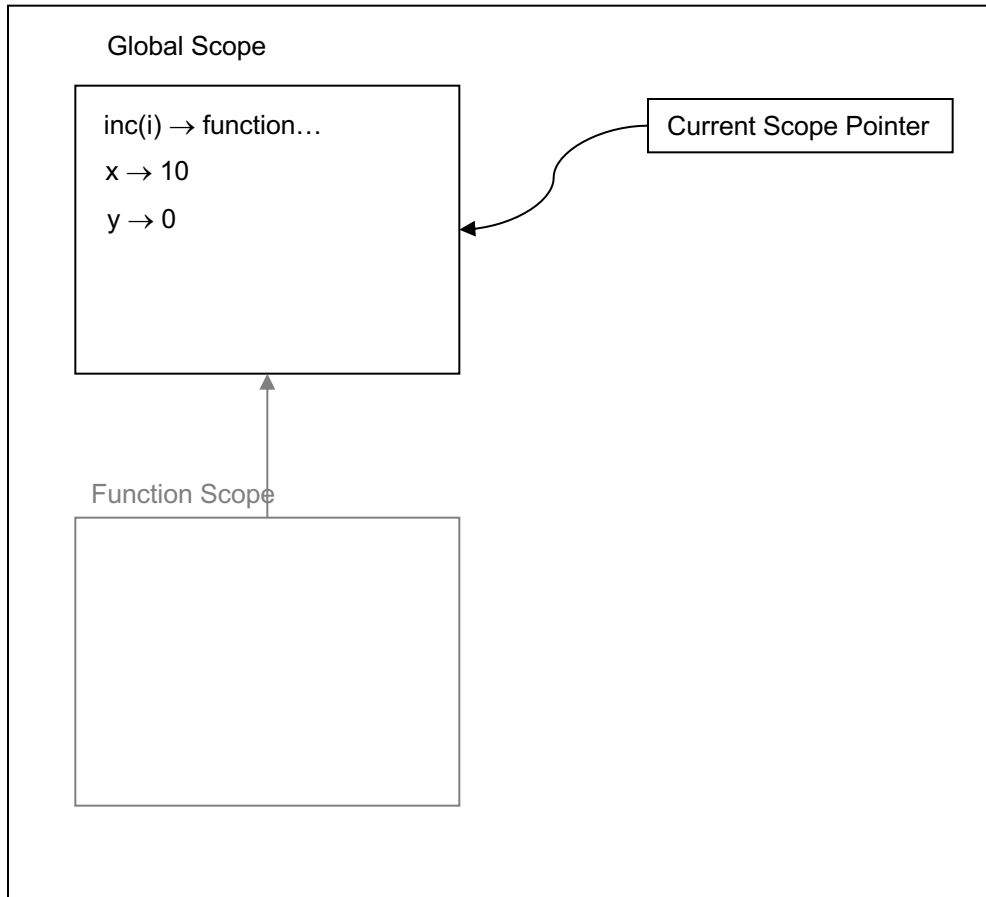



→ declare inc(i) return i+1;  
declare x = 10;  
declare y;  
y = inc(x);  
put y;

# Programs with Functions



Symbol Table

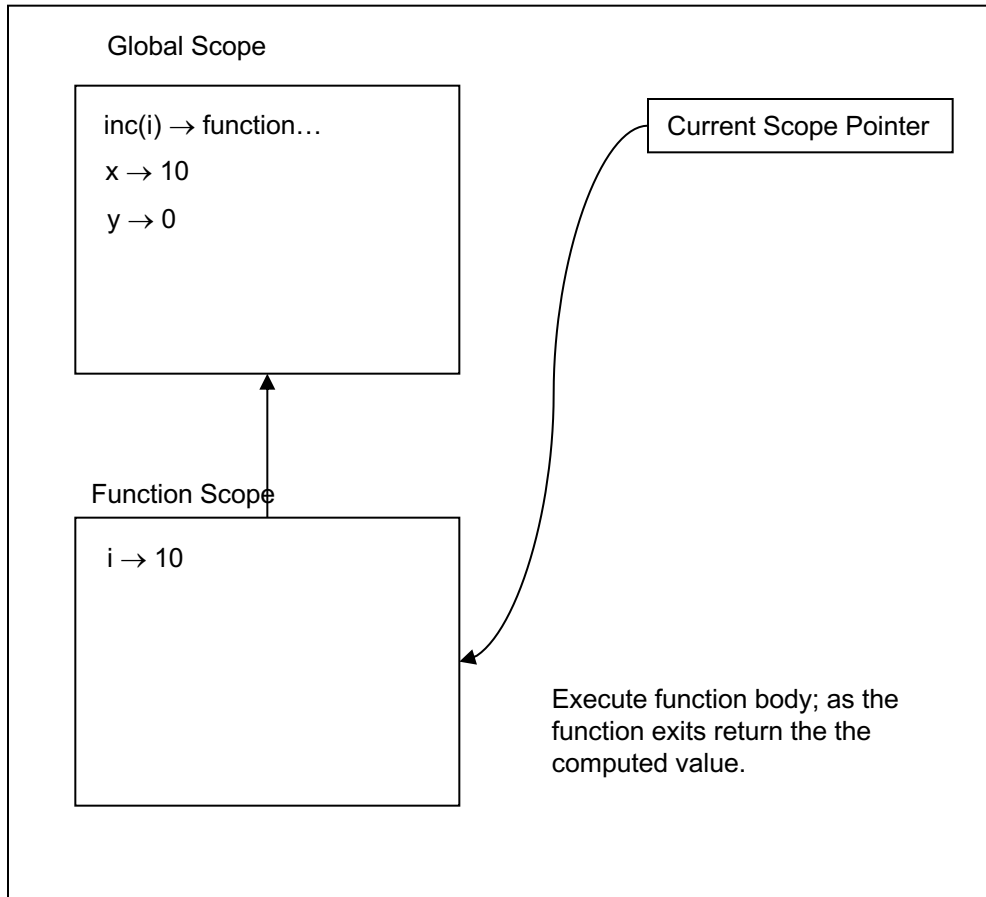


 `declare inc(i) return i+1;`  
`declare x = 10;`  
`declare y;`  
`y = inc(x);`  
`put y;`

# Programs with Functions



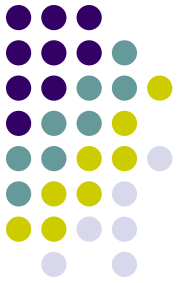
Symbol Table



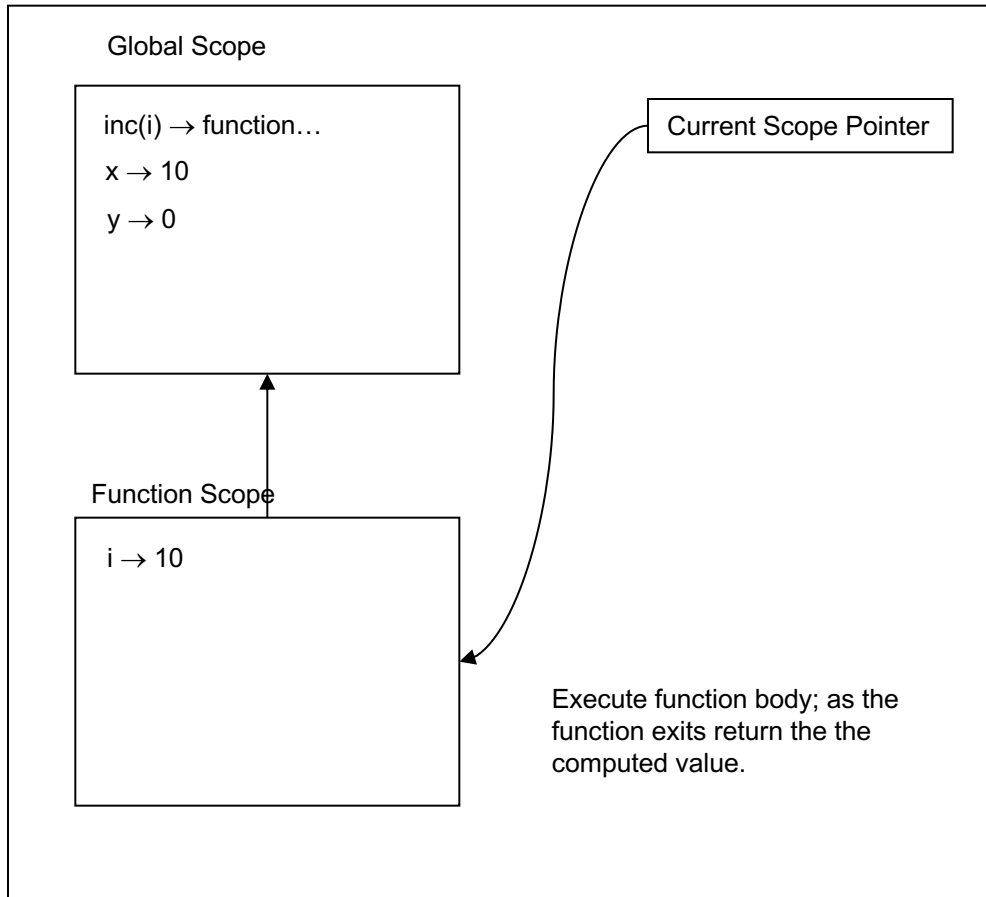
declare inc(i) return i+1;  
  
declare x = 10;  
declare y;  
y = inc(x);  
put y;



# Programs with Functions

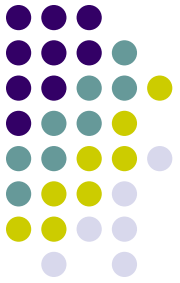


Symbol Table

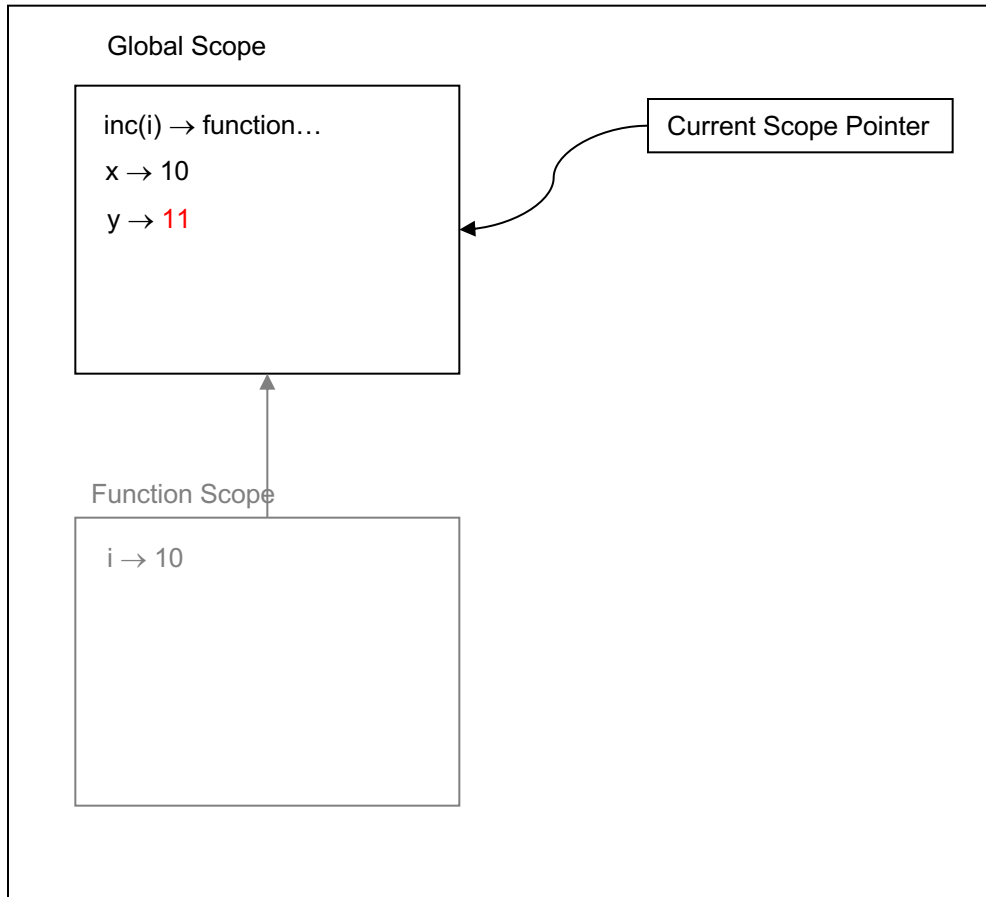



→ declare inc(i) return i+1;  
declare x = 10;  
declare y;  
y = inc(x);  
put y;

# Programs with Functions



Symbol Table

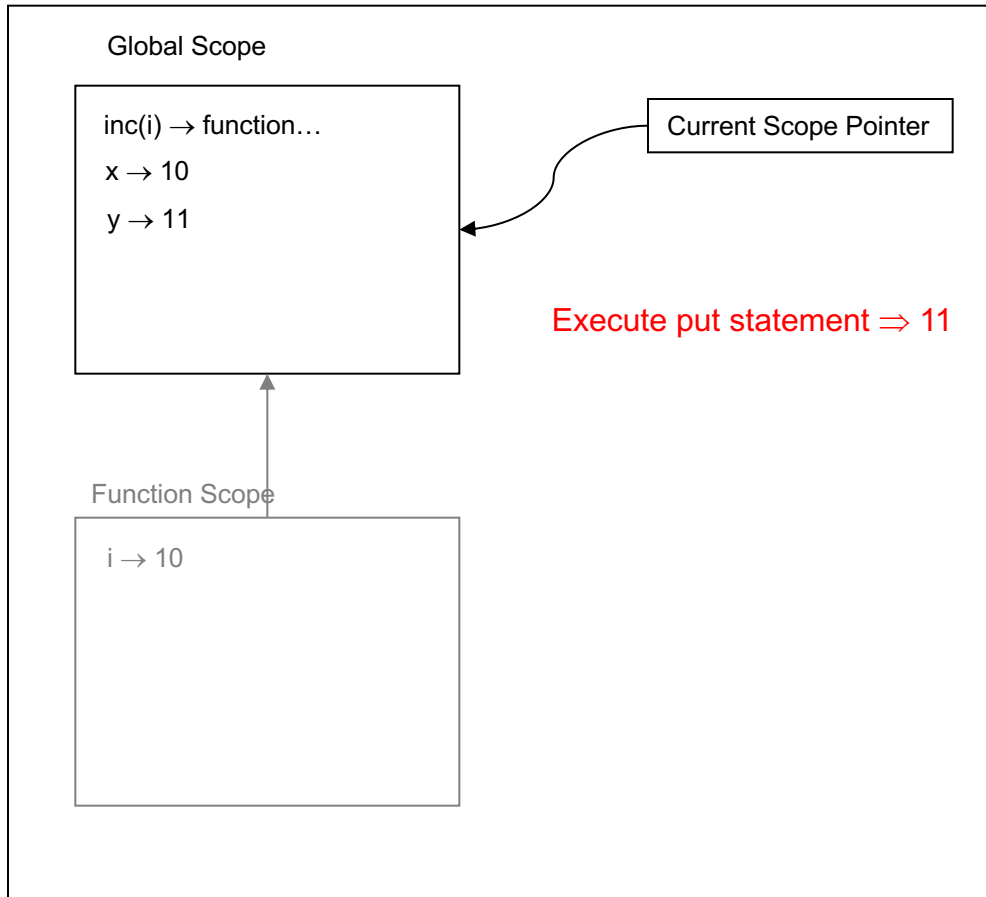


 `declare inc(i) return i+1;`  
`declare x = 10;`  
`declare y;`  
`y = inc(x);`  
`put y;`

# Programs with Functions



Symbol Table

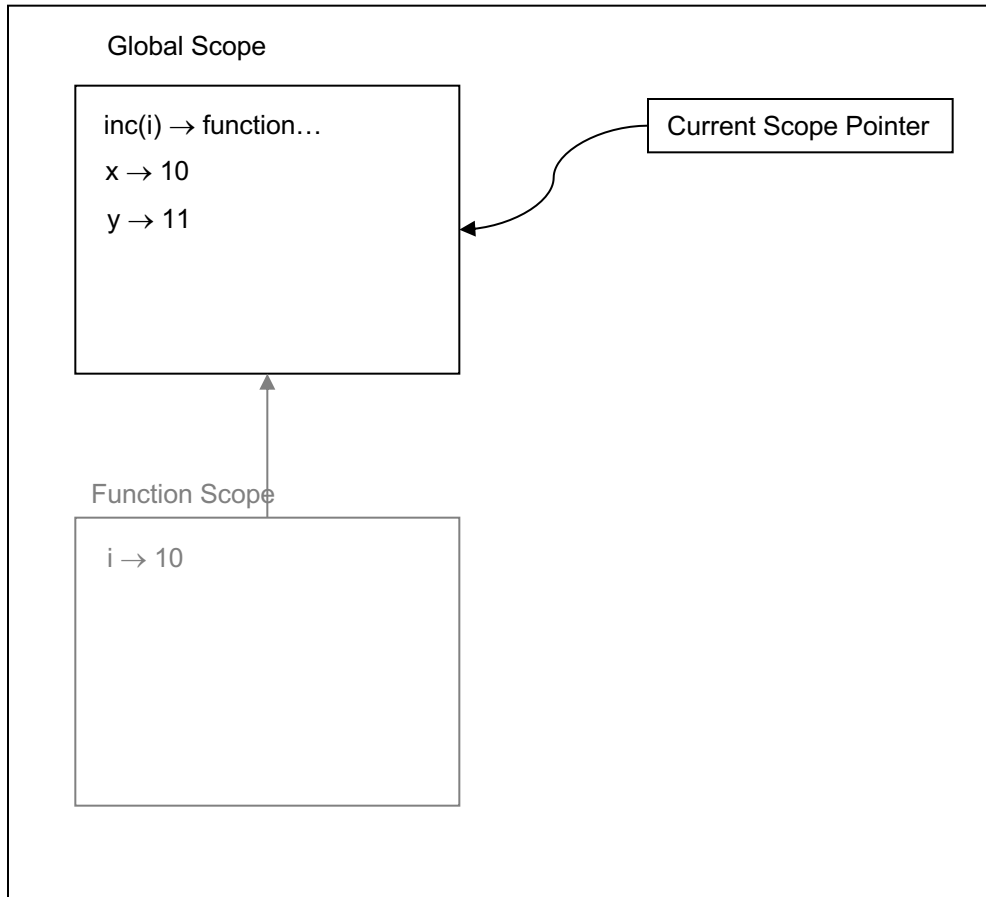


declare inc(i) return i+1;  
  
declare x = 10;  
declare y;  
y = inc(x);  
put y;

# Programs with Functions



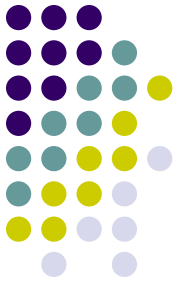
Symbol Table



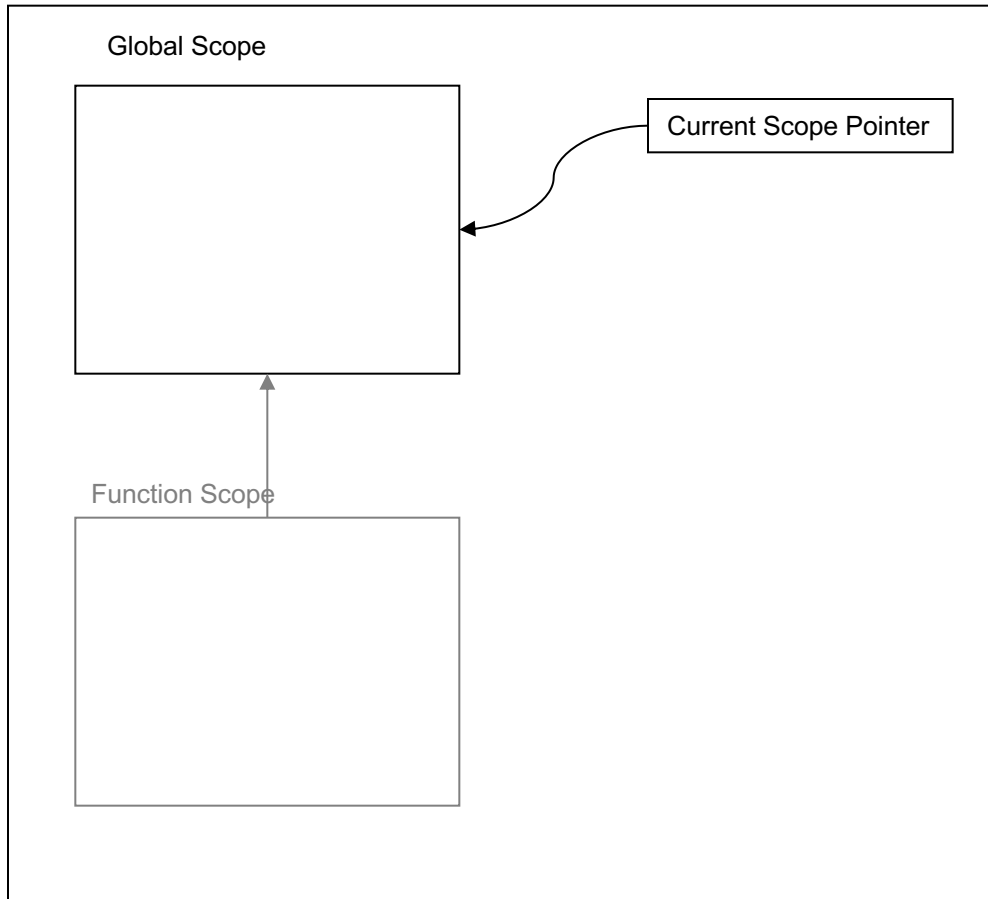
```
declare inc(i) return i+1;
```

```
declare x = 10;  
declare y;  
y = inc(x);  
put y;
```

# Multiple Parameters

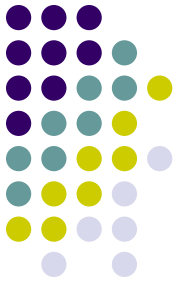


Symbol Table

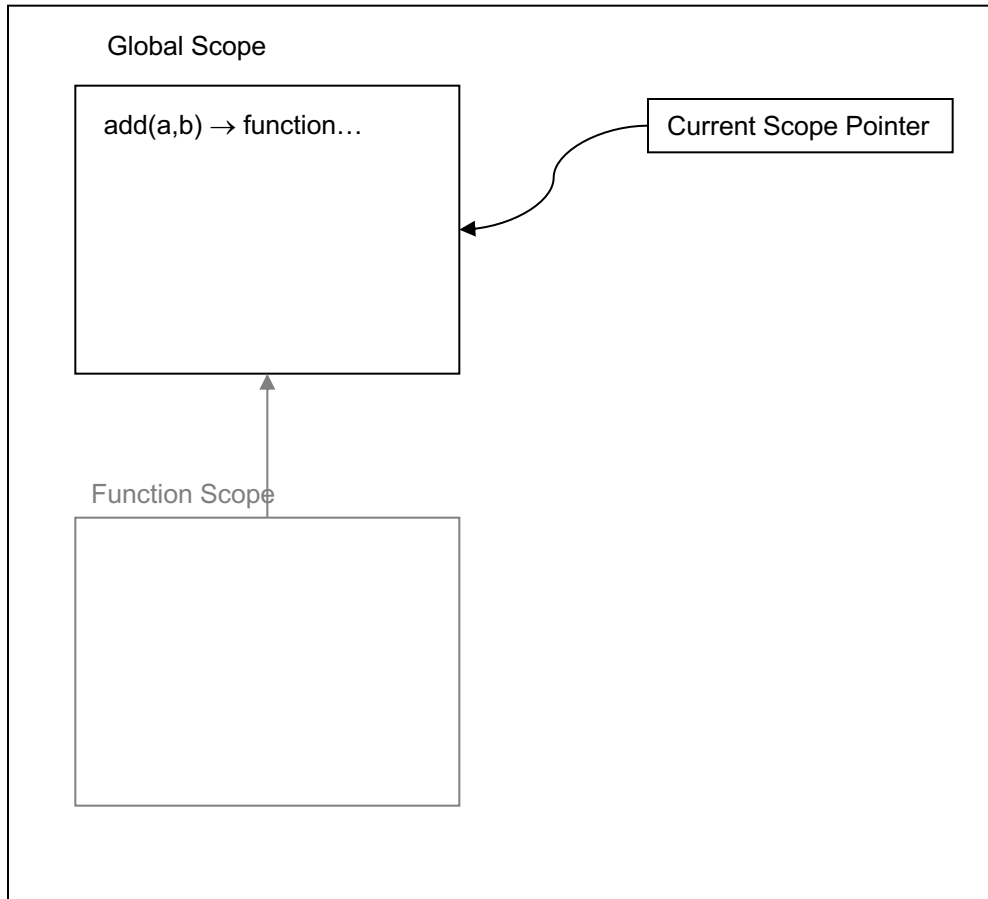


```
declare add(a,b) return a+b;  
put add(3,2);
```

# Multiple Parameters



Symbol Table

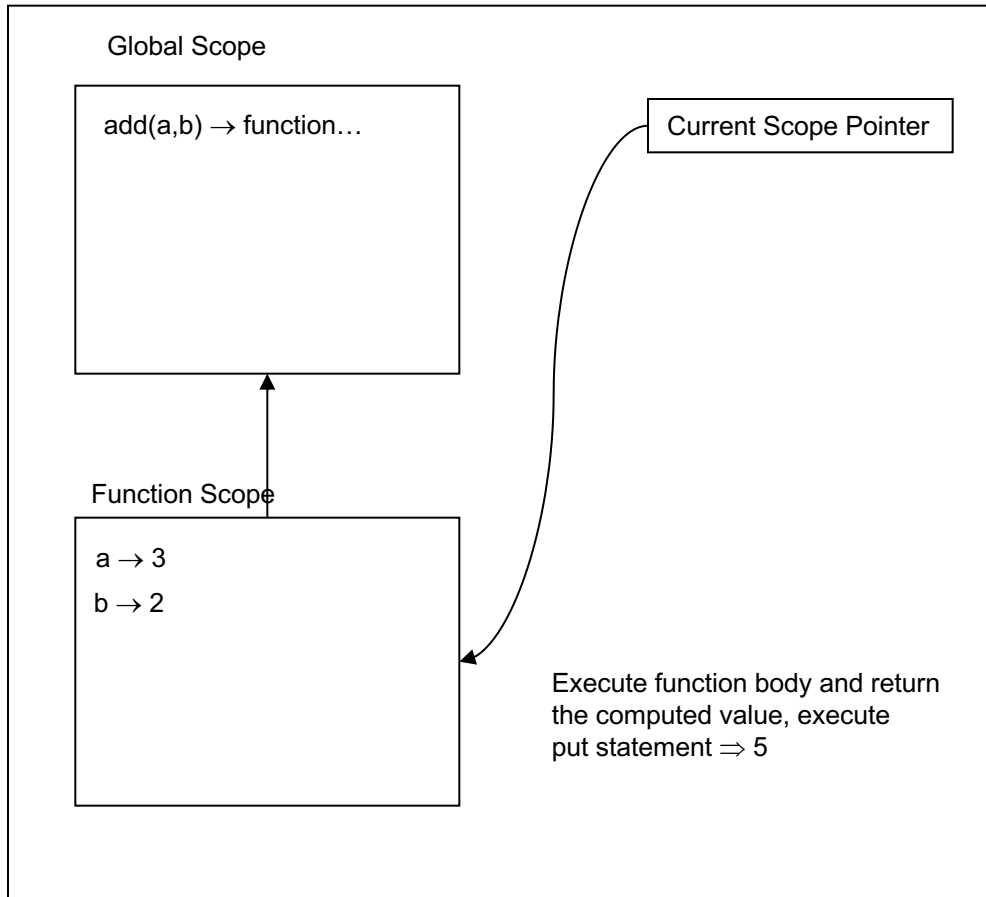


→ declare add(a,b) return a+b;  
put add(3,2);

# Multiple Parameters



Symbol Table

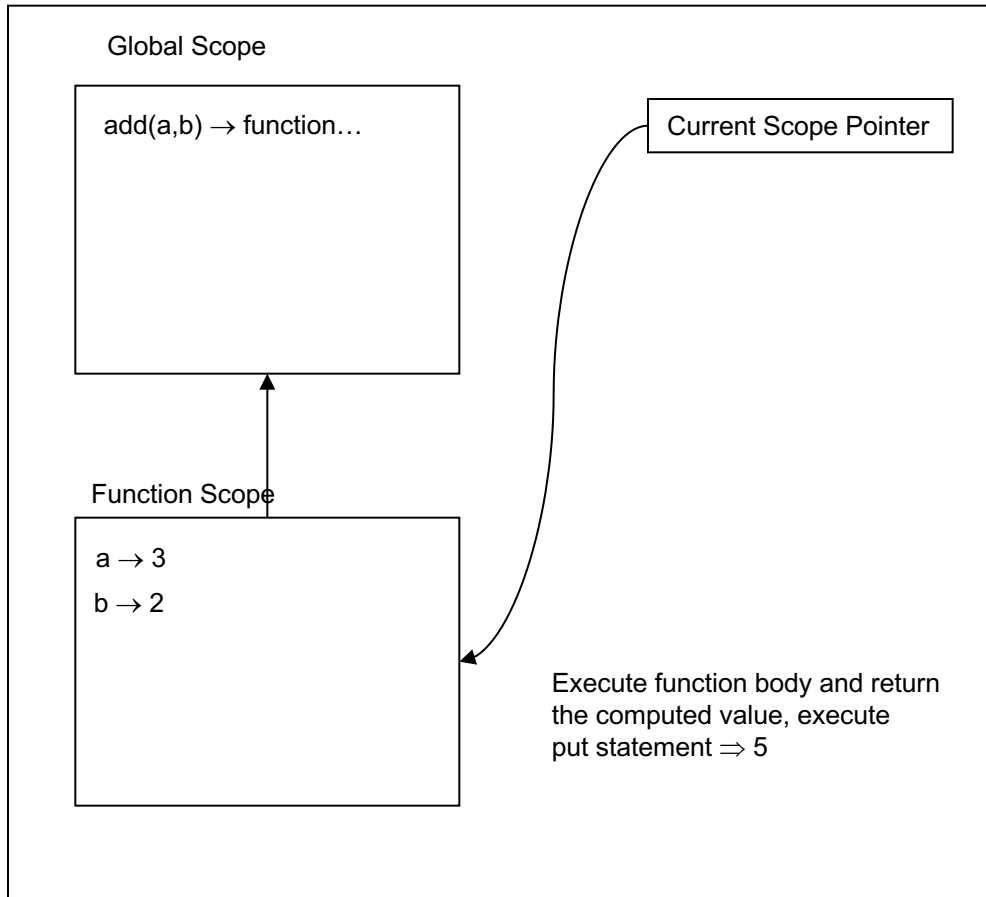


declare add(a,b) return a+b;  
→ put add(3,2);

# Multiple Parameters



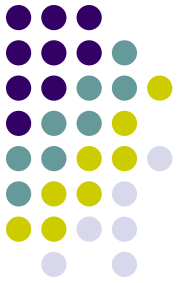
Symbol Table



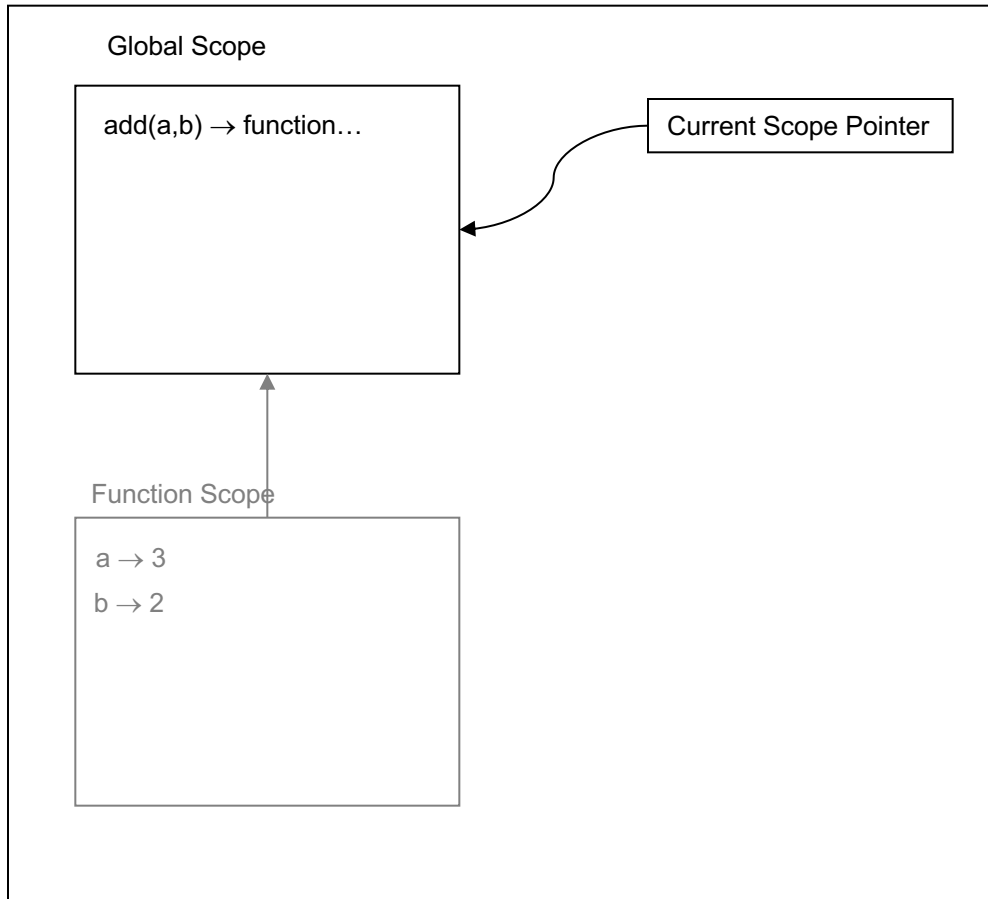
→ declare add(a,b) return a+b;  
put add(3,2);



# Multiple Parameters



Symbol Table

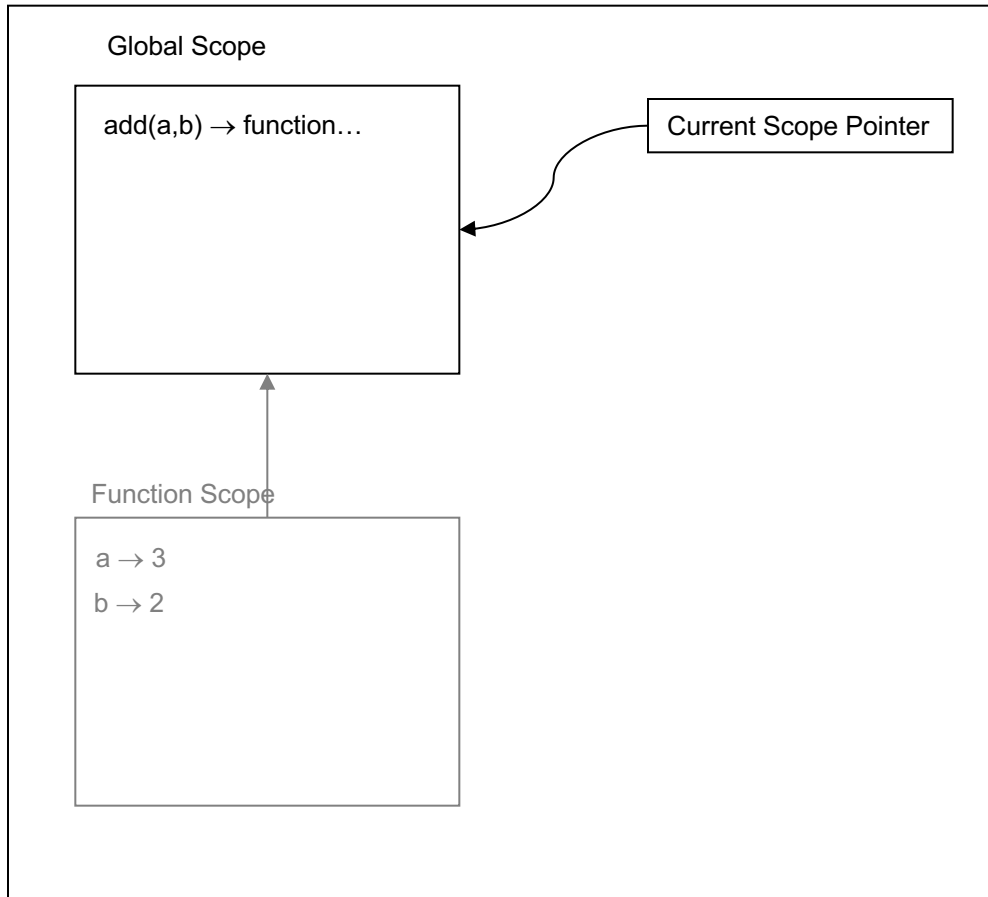


declare add(a,b) return a+b;  
→ put add(3,2);

# Multiple Parameters



Symbol Table

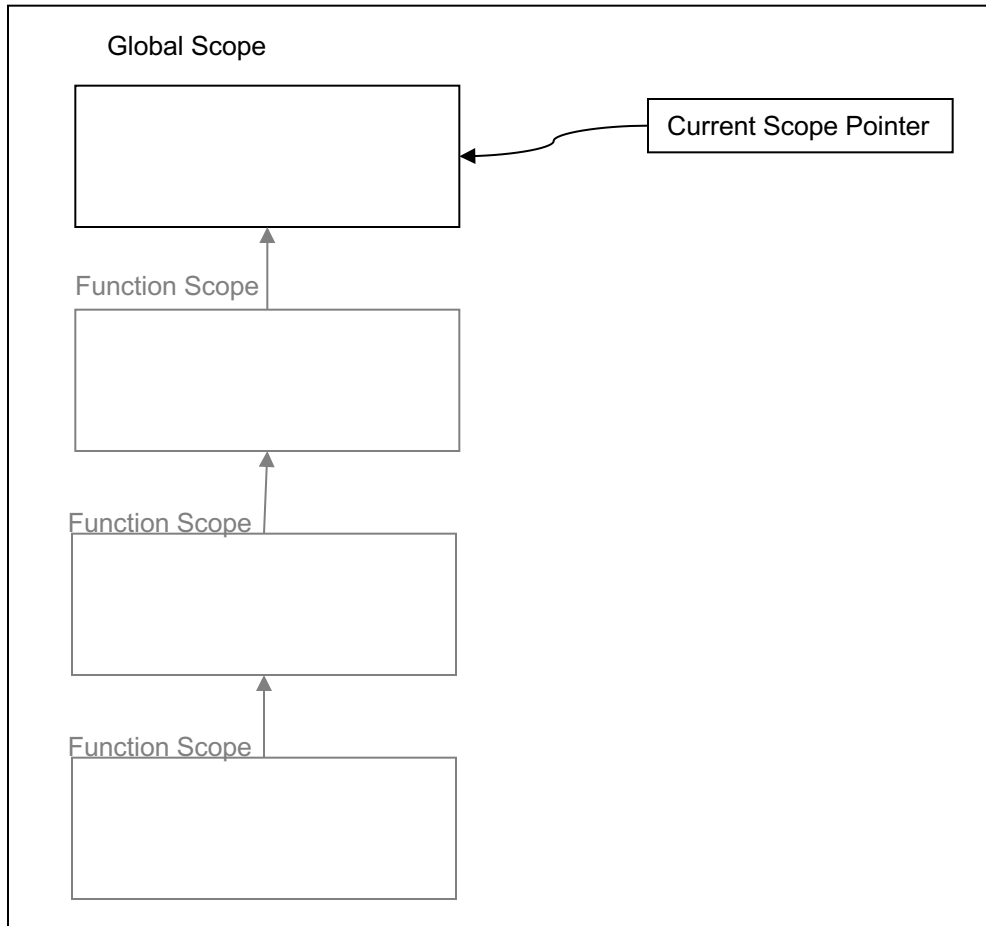


```
declare add(a,b) return a+b;  
put add(3,2);
```

# Recursion

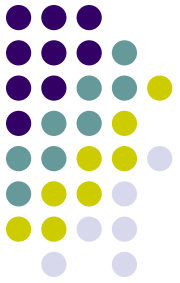


Symbol Table

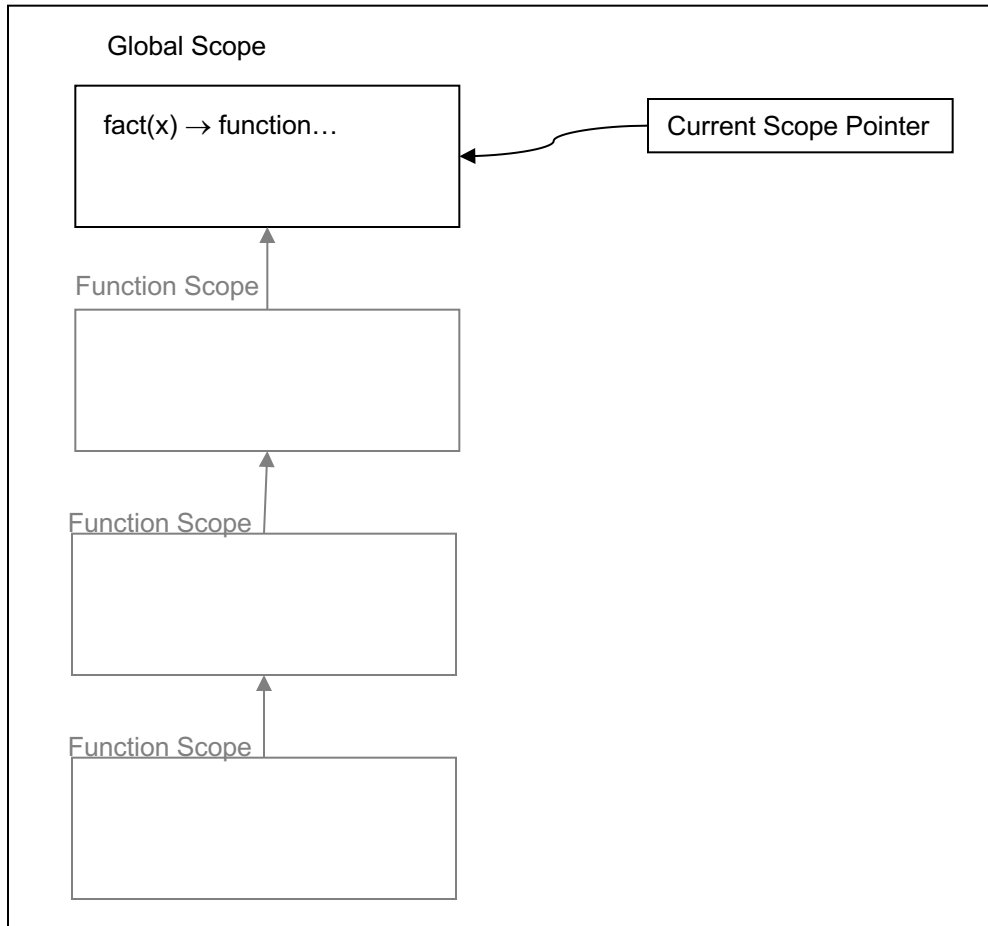


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion

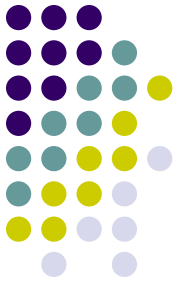


Symbol Table

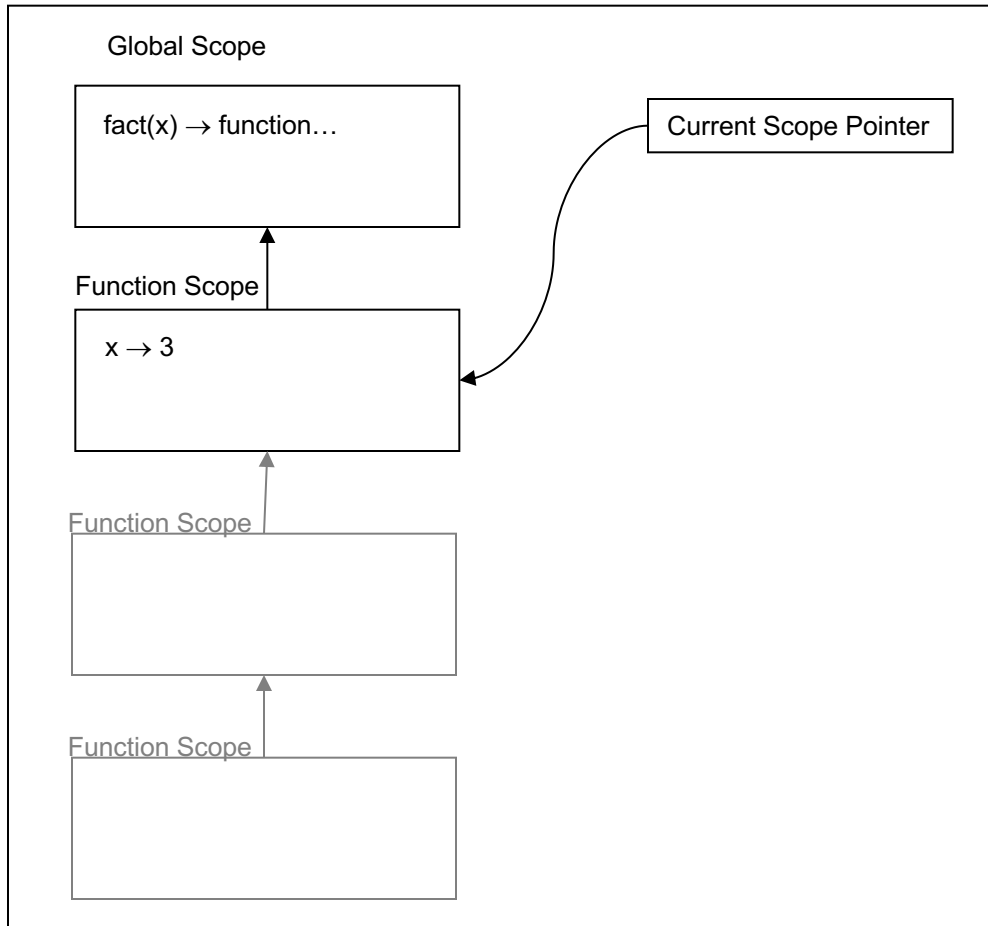


→ declare fact(x) {  
    declare y;  
    if (x <= 1)  
        return 1;  
    else {  
        y = x\*fact(x-1);  
        return y;  
    }  
}  
  
put fact(3);

# Recursion



Symbol Table

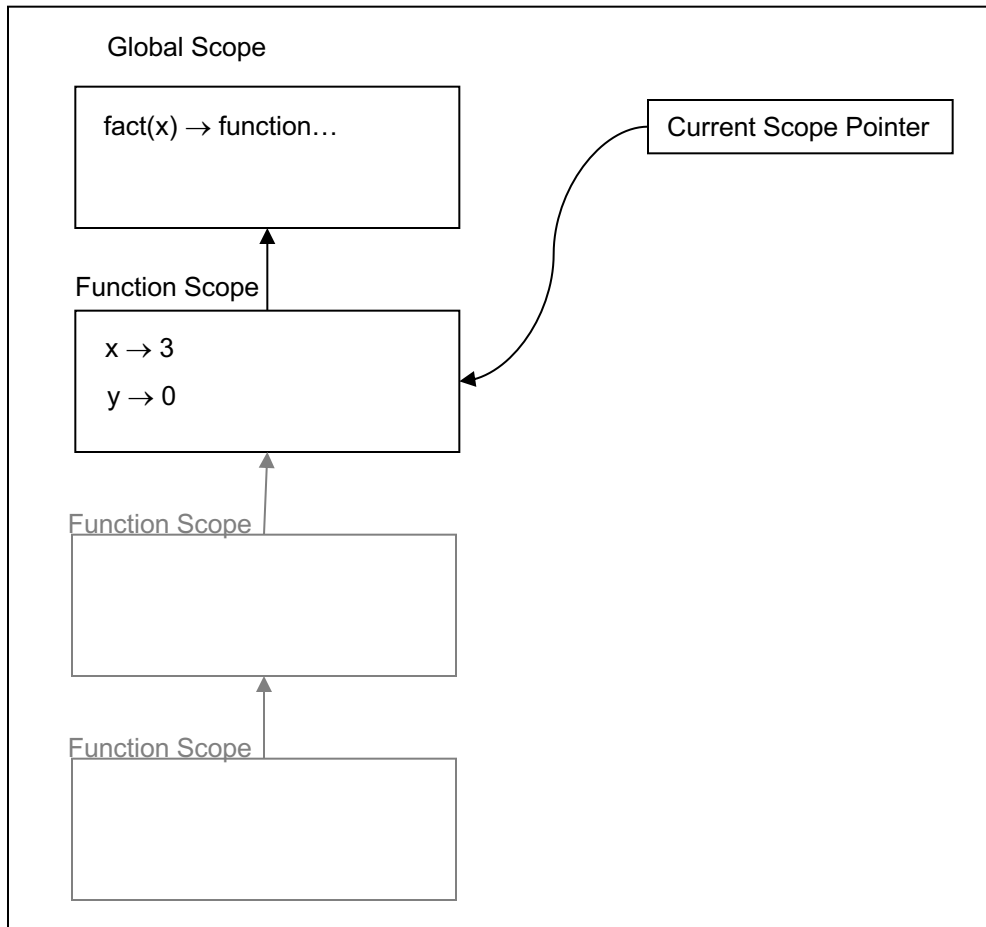


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
→ put fact(3);
```

# Recursion



Symbol Table

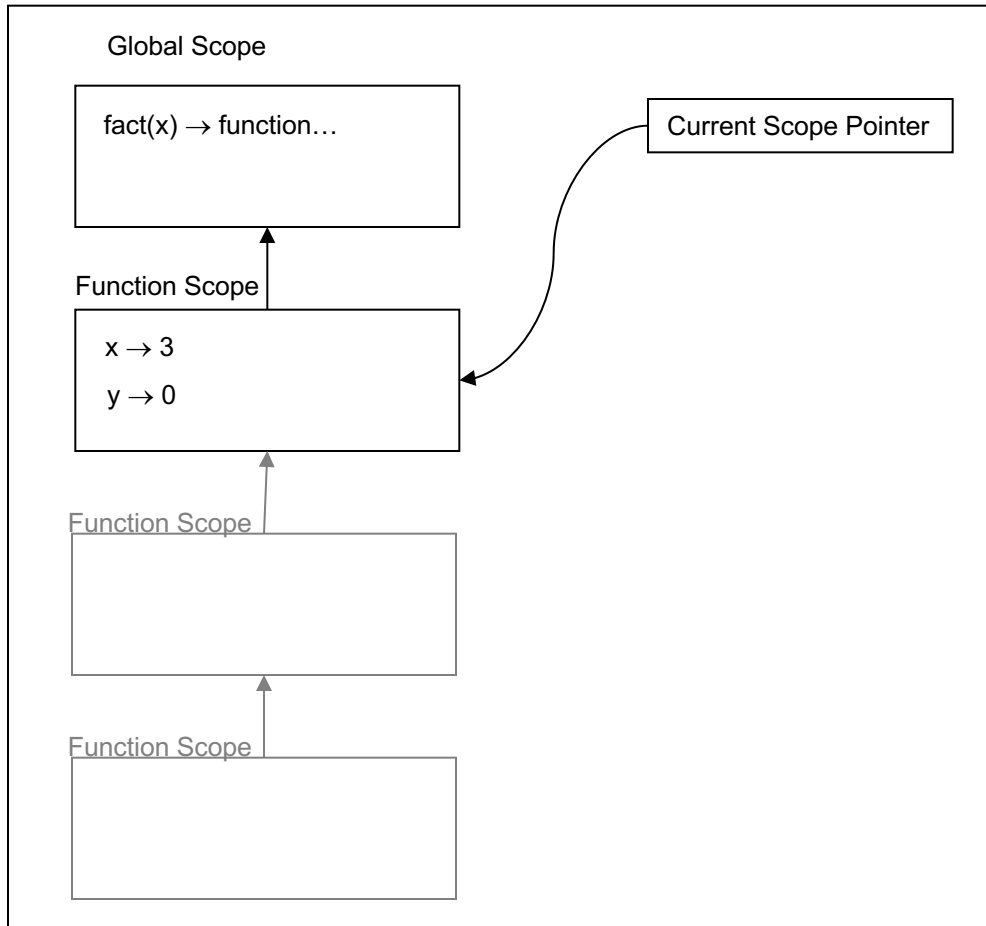


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion

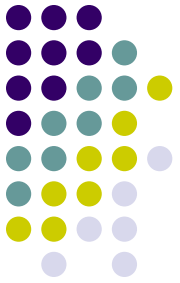


Symbol Table

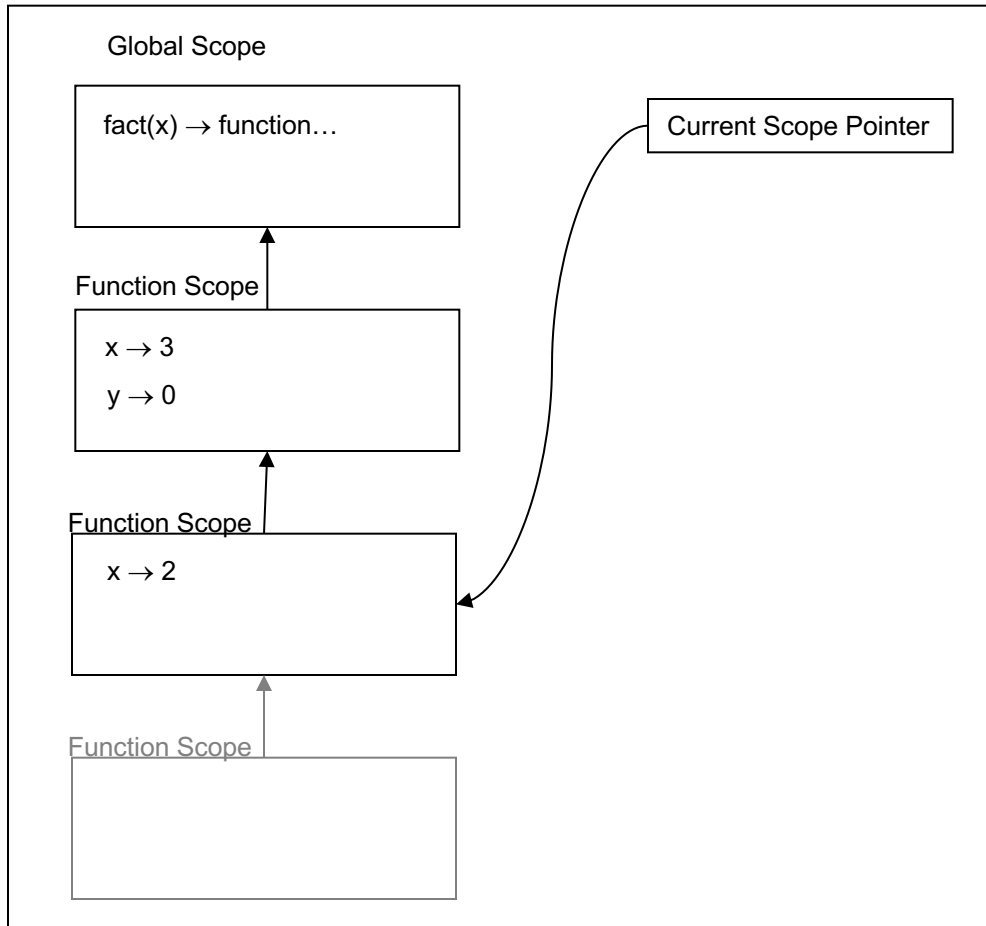


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion



Symbol Table

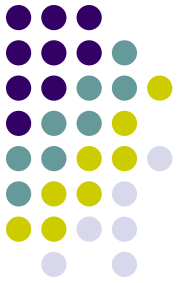


→

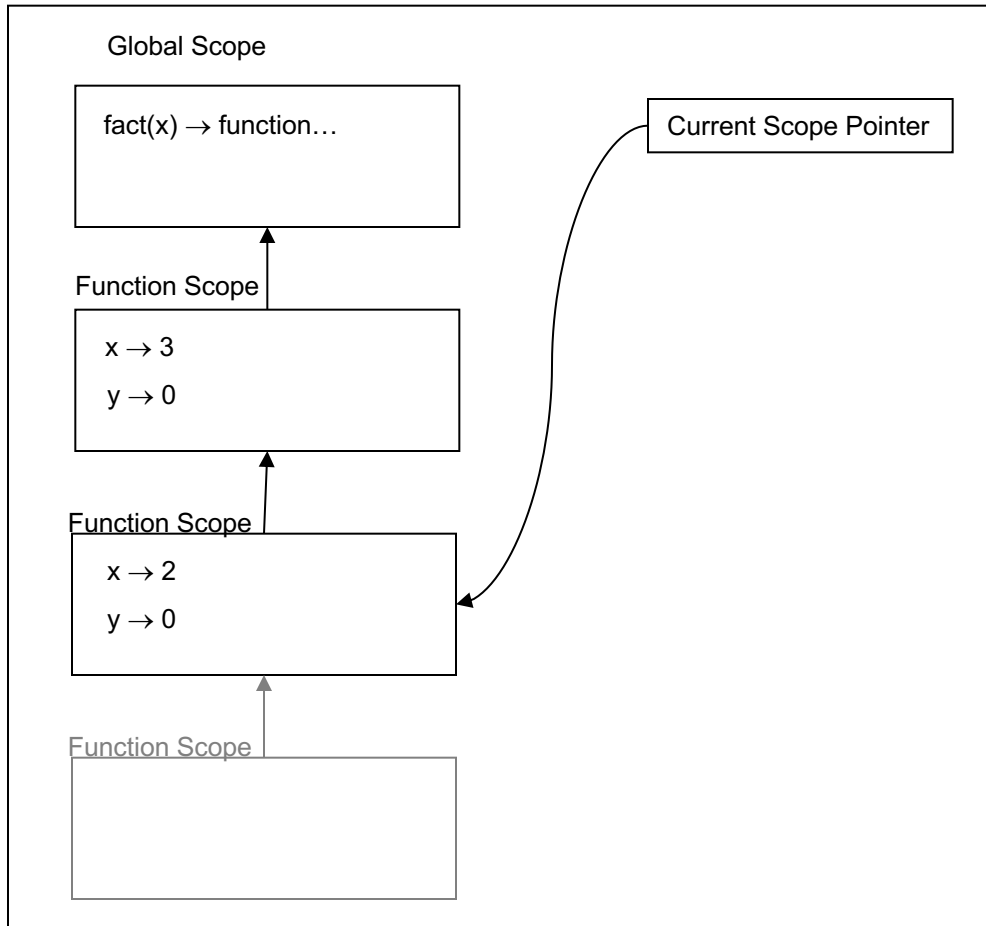
```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```



# Recursion

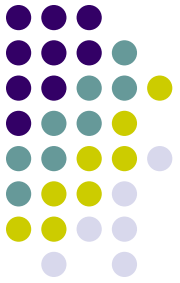


Symbol Table


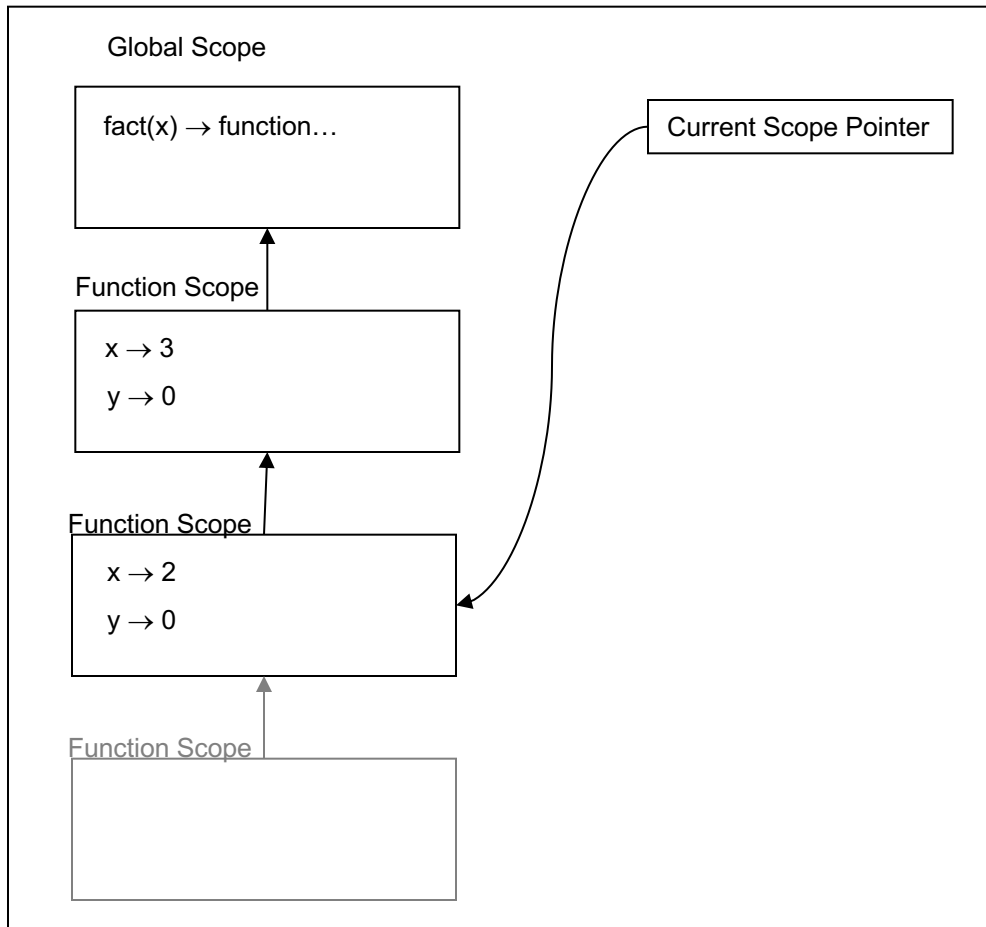


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion



Symbol Table


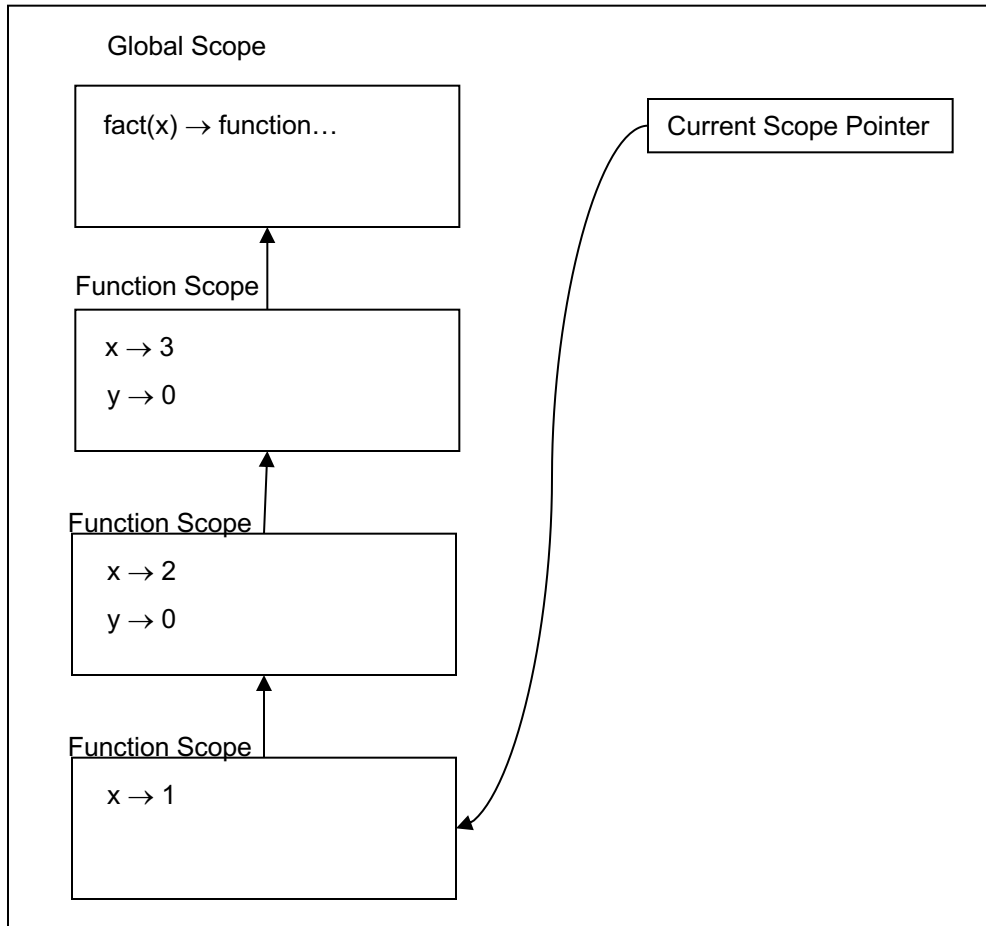


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion



Symbol Table

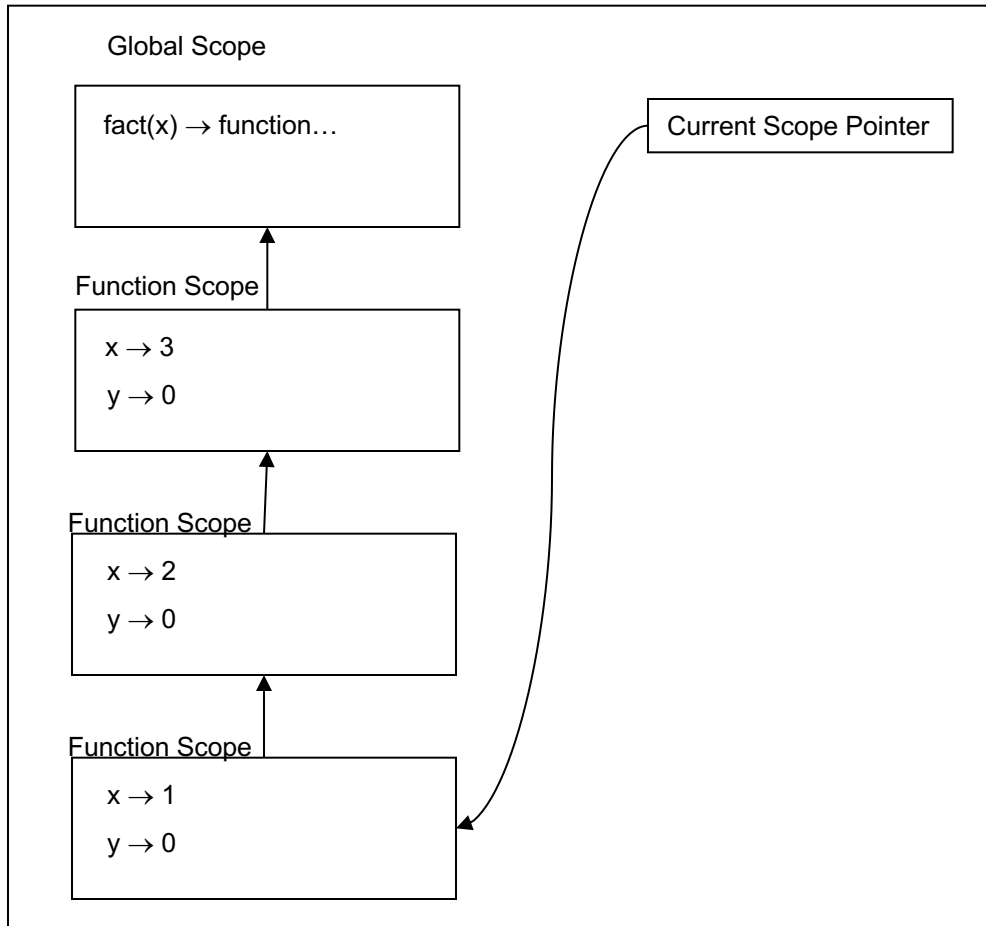


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion



Symbol Table

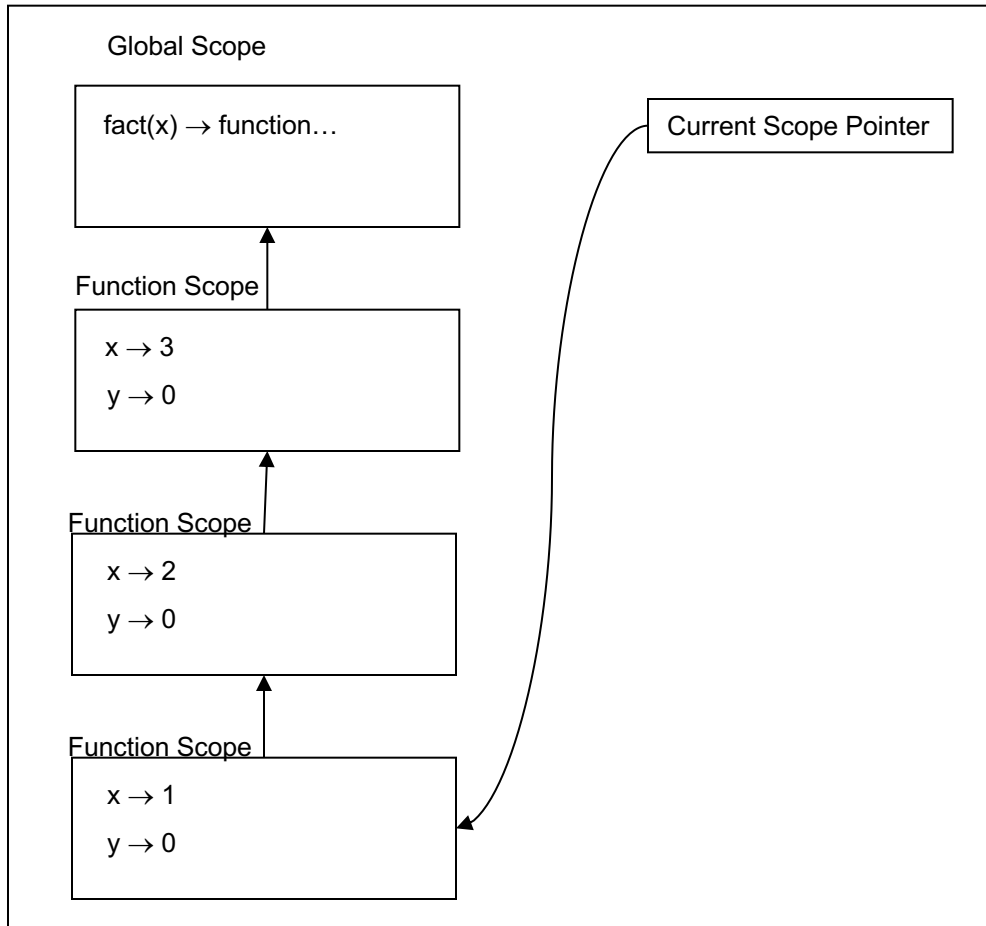


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion

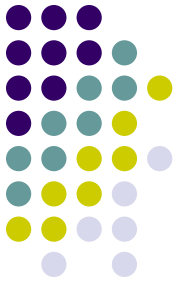


Symbol Table

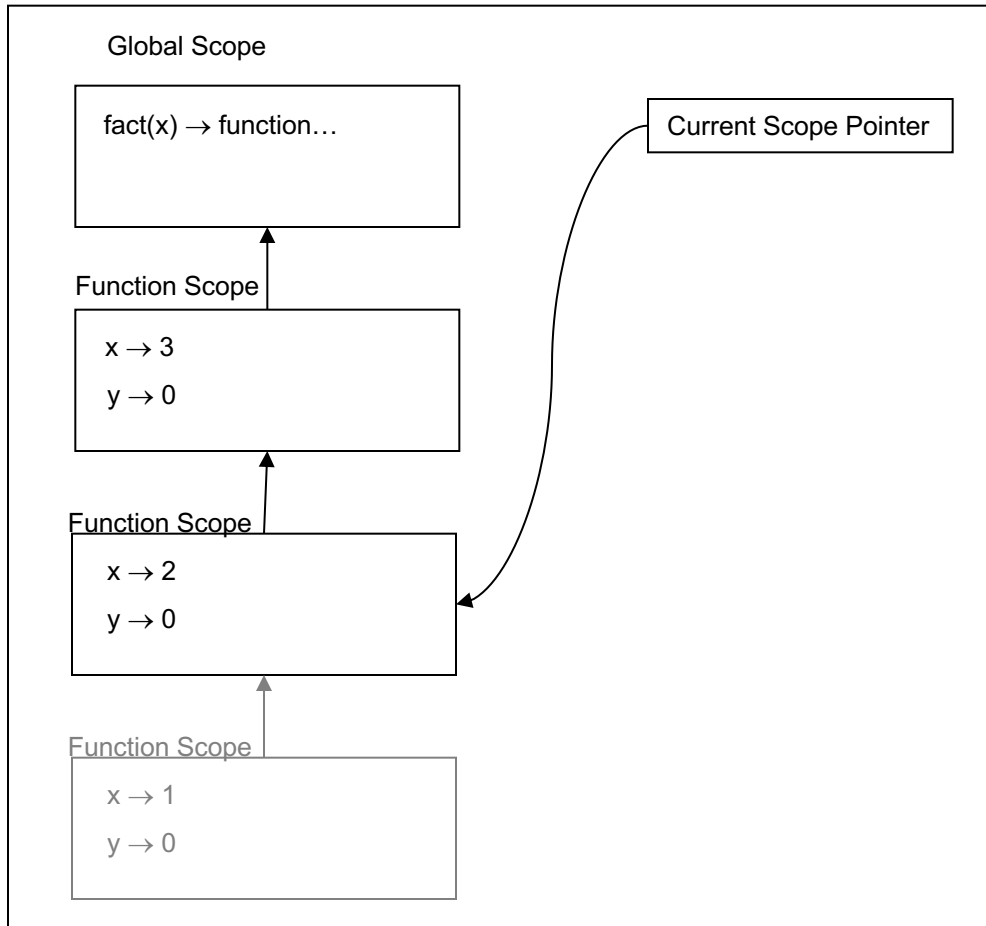


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion



Symbol Table




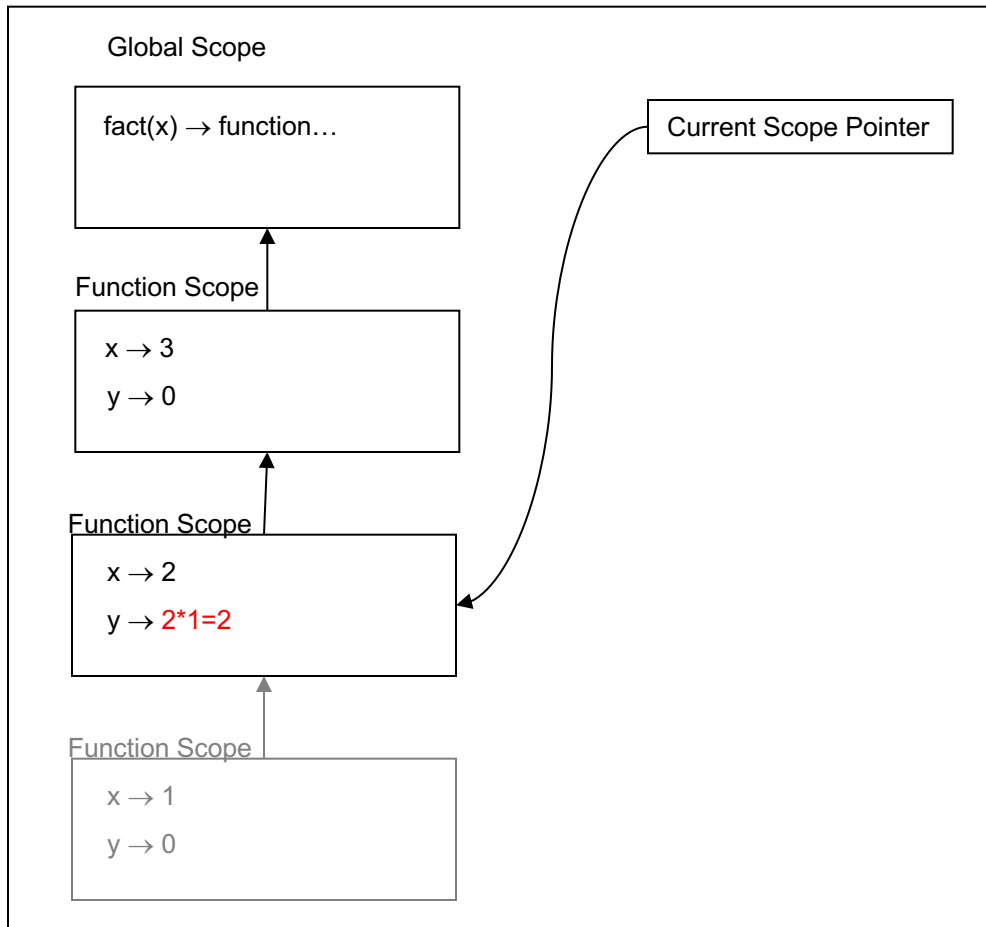
→

```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion



Symbol Table


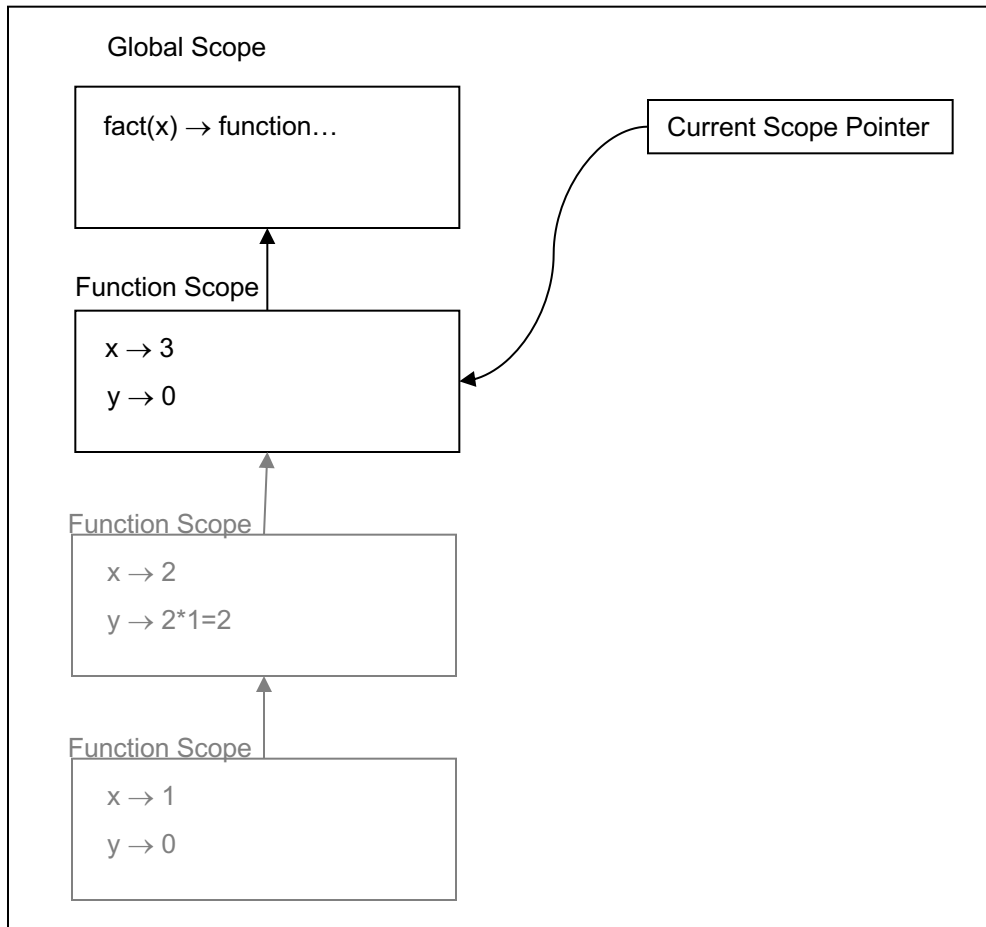


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion



Symbol Table




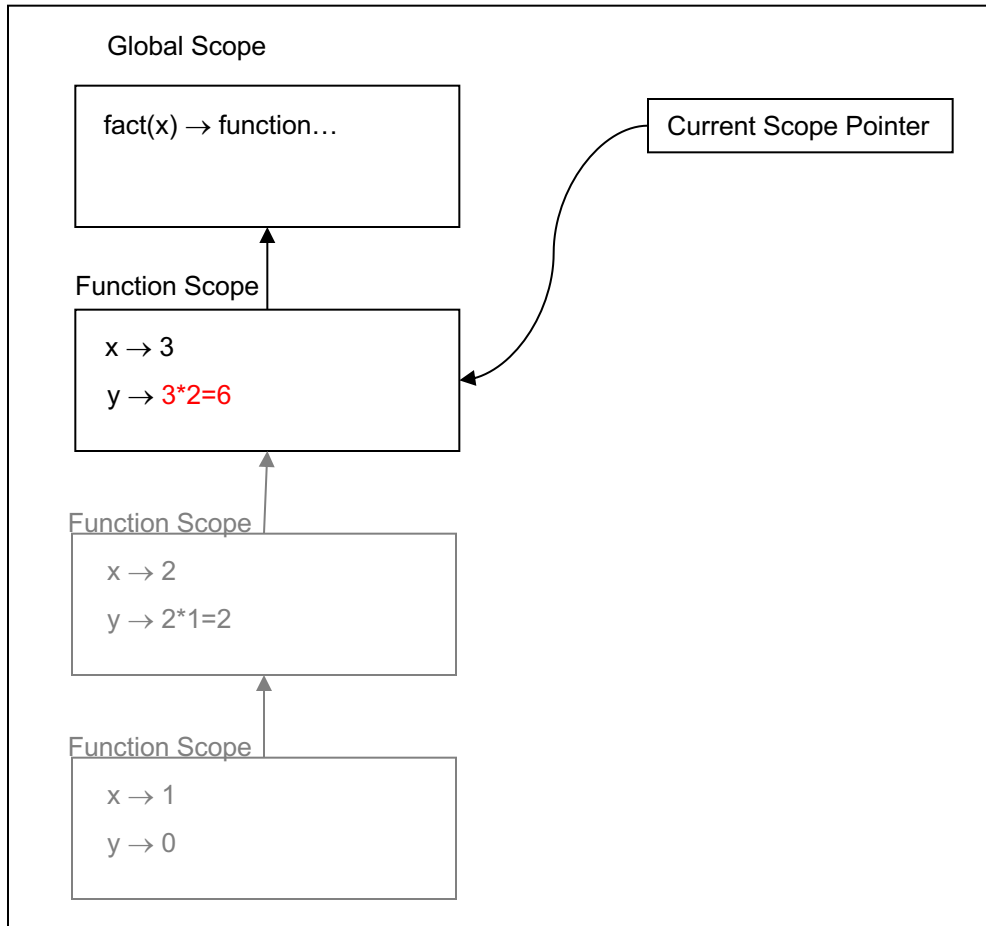
```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```



# Recursion



Symbol Table




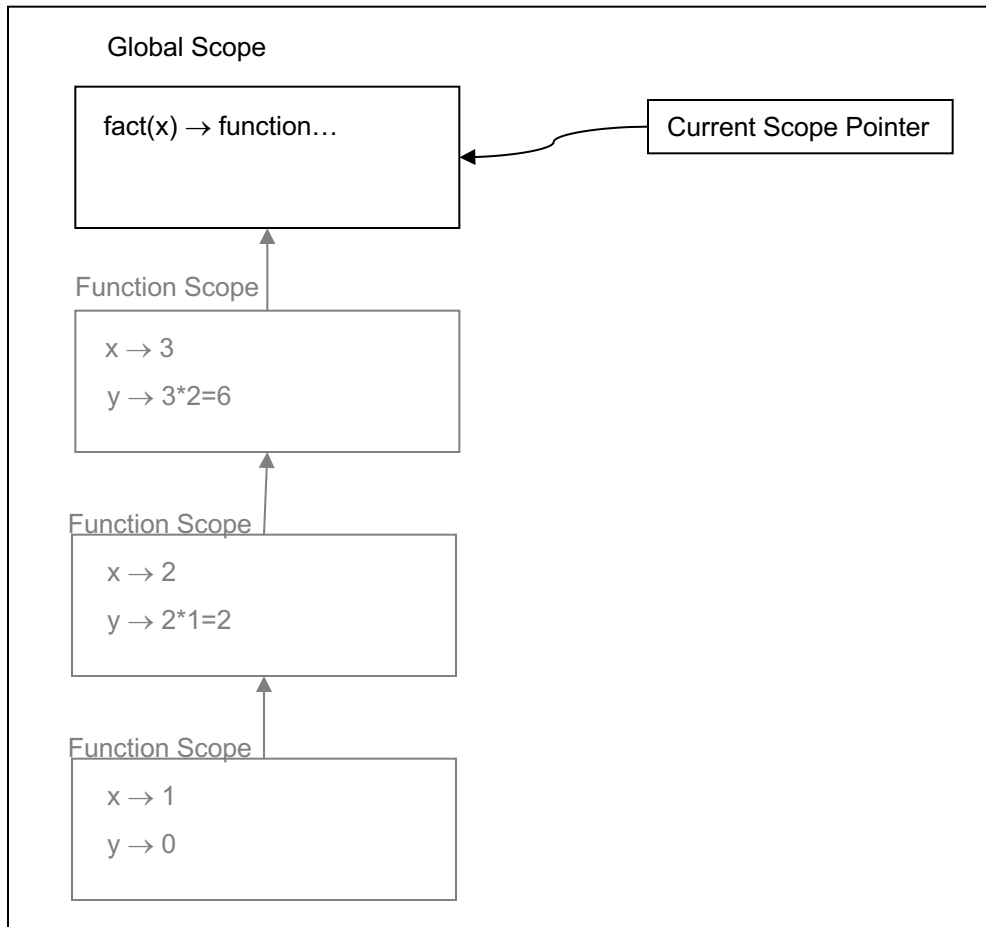
```
declare fact(x) {
  declare y;
  if (x <= 1)
    return 1;
  else {
    y = x*fact(x-1);
    return y;
  }
}

put fact(3);
```

# Recursion



Symbol Table

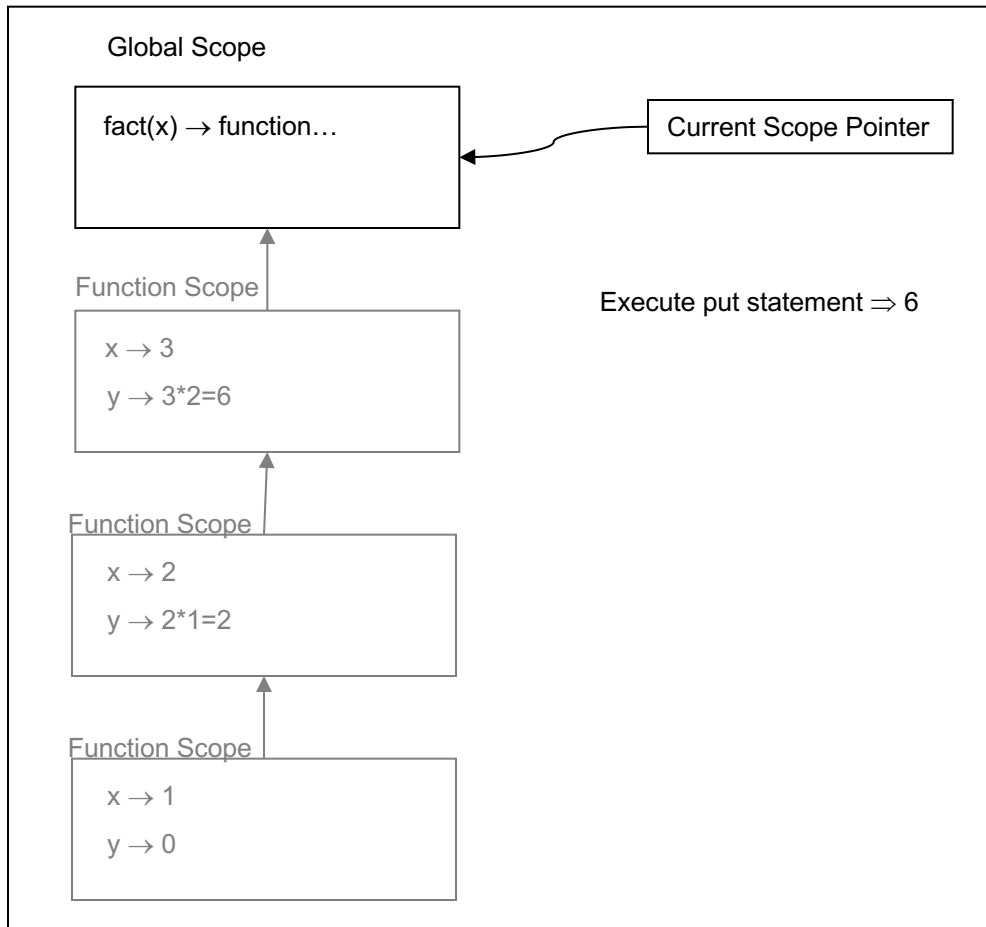


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```

# Recursion

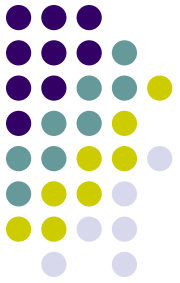


Symbol Table

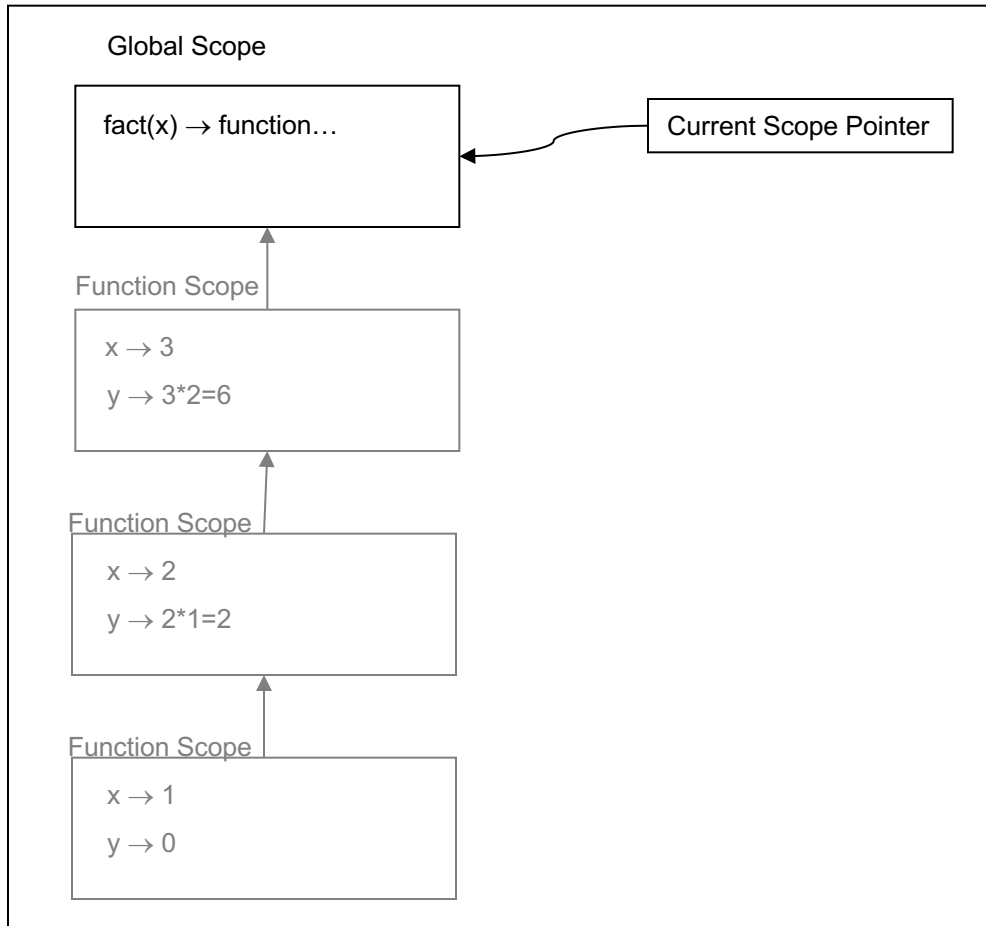


```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
→ put fact(3);
```

# Recursion



Symbol Table



```
declare fact(x) {  
  declare y;  
  if (x <= 1)  
    return 1;  
  else {  
    y = x*fact(x-1);  
    return y;  
  }  
}  
  
put fact(3);
```



# Static vs. Dynamic Scoping

- There is an interesting interaction between function scopes and global variables
- Consider the following program:

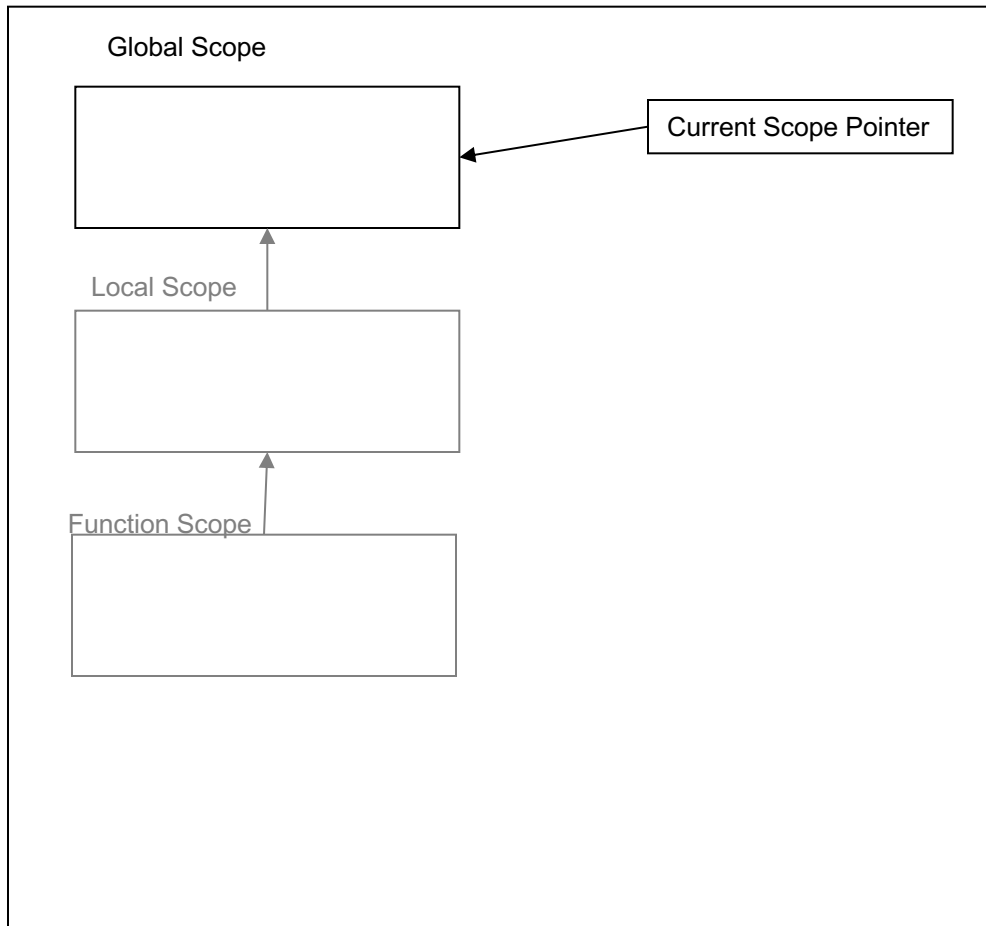
```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

What is the expected output of the program?

# Dynamic Scoping



Symbol Table



```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

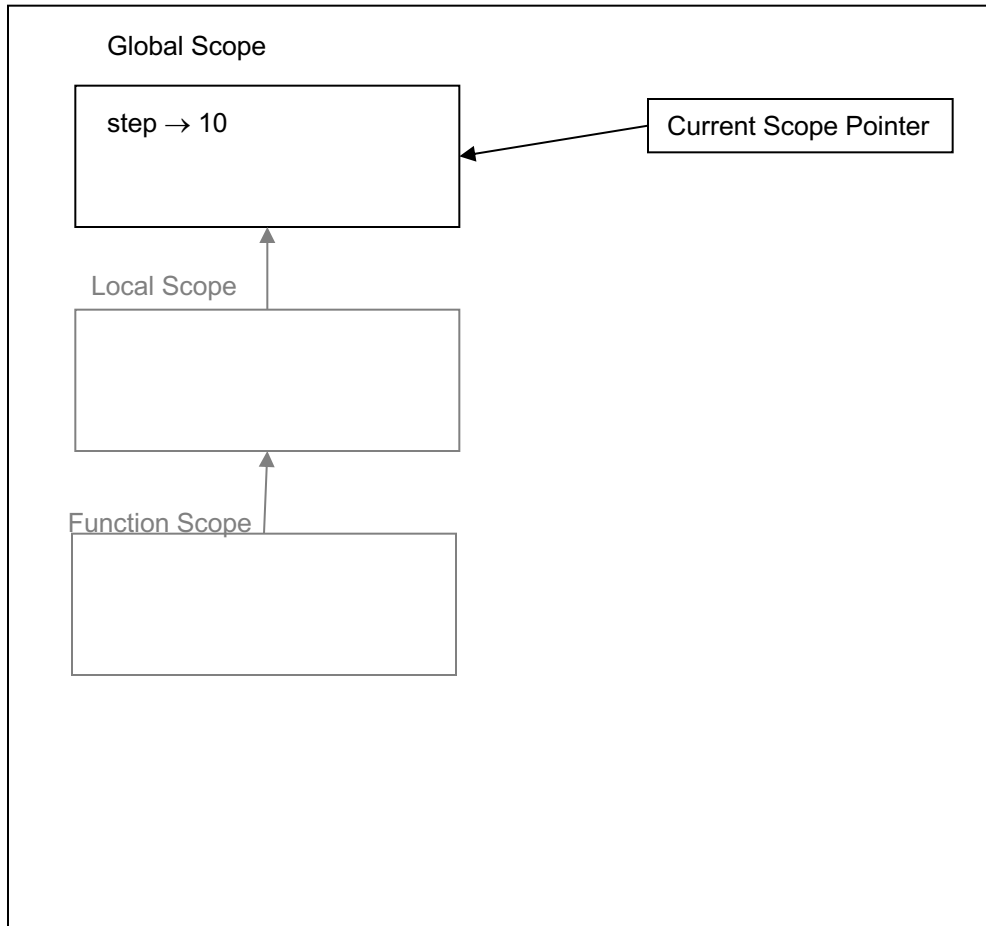
☞ In dynamic scoping we treat the function scope like any other local scope and push it on the scope stack.

What is the expected output of the program under these assumptions?

# Dynamic Scoping

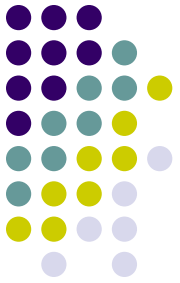


Symbol Table

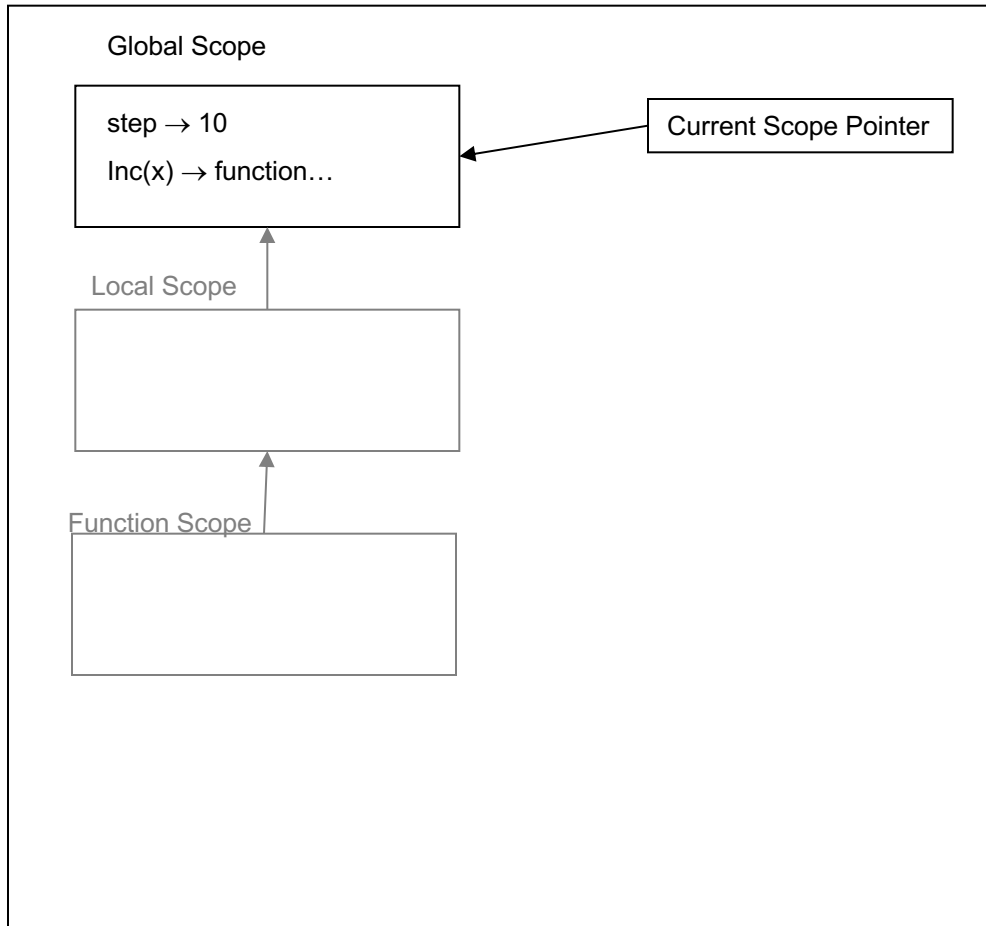


→ declare step = 10;  
declare inc(x) {  
    return x+step;  
}  
// start a local scope...  
{  
    declare step = 2;  
    put inc(5);  
}

# Dynamic Scoping



Symbol Table



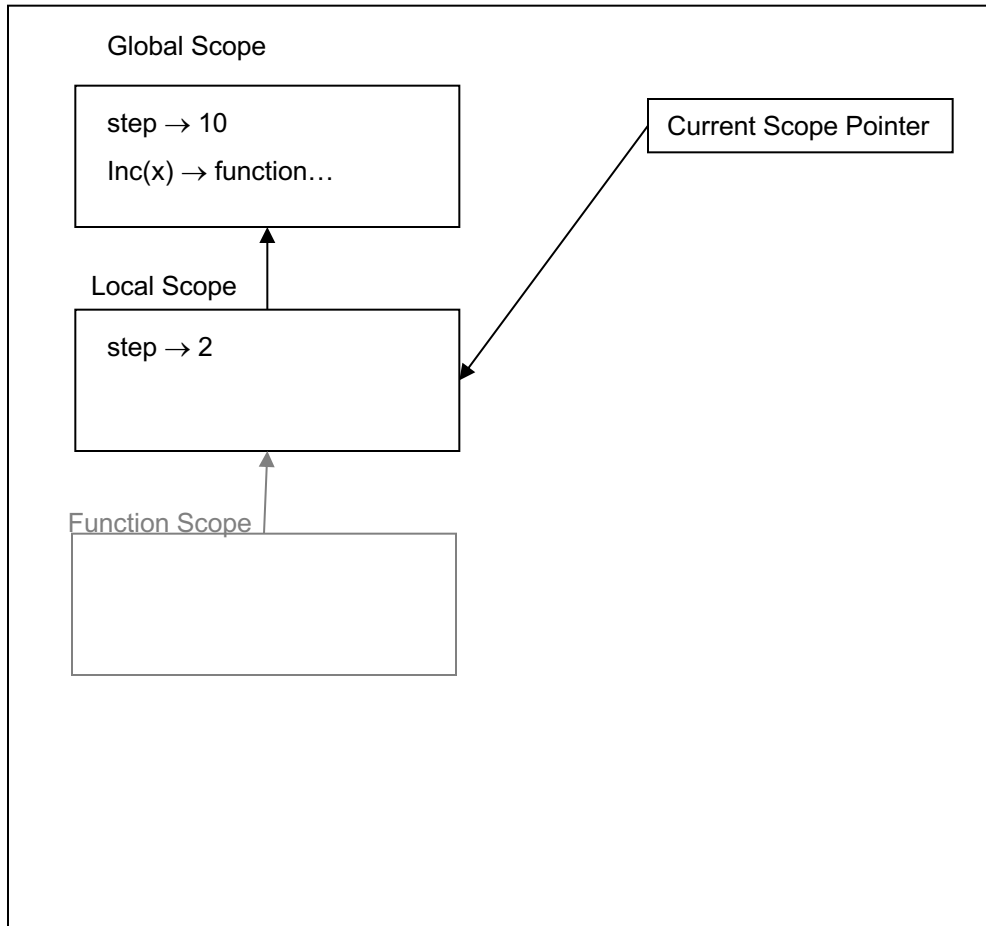
```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```



# Dynamic Scoping



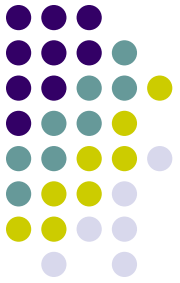
Symbol Table



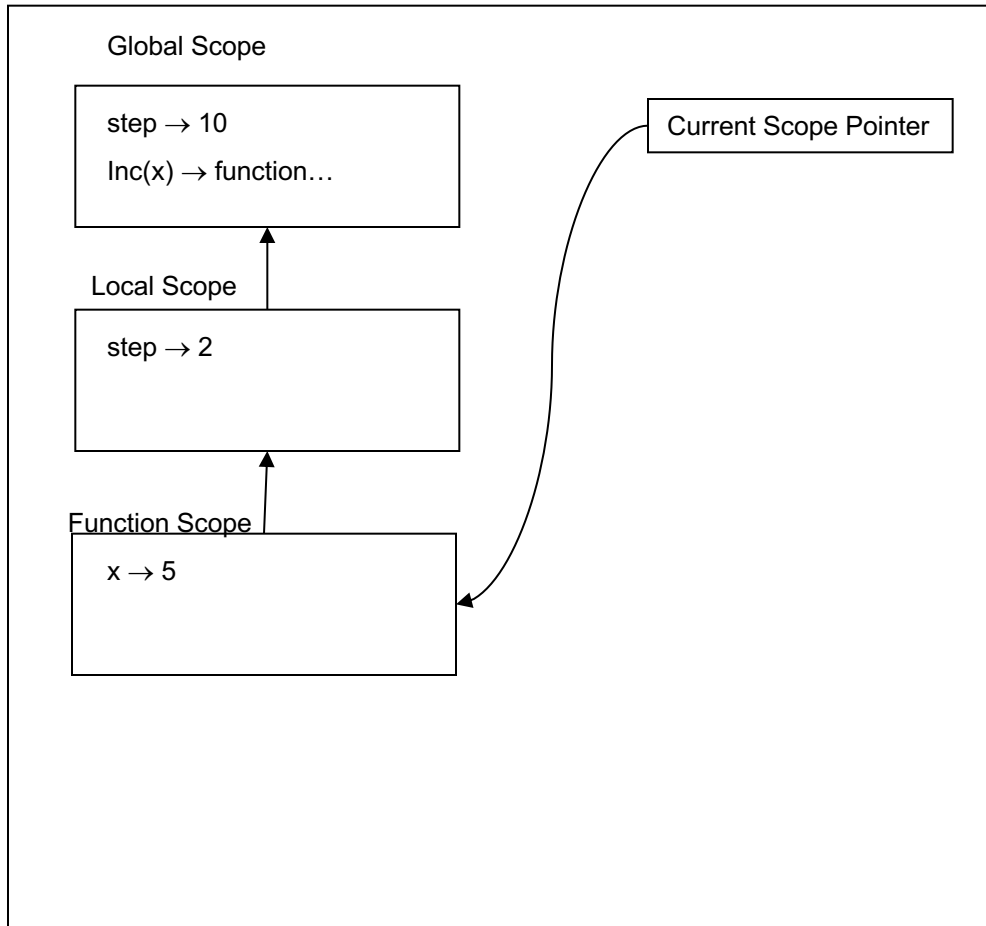
→

```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Dynamic Scoping



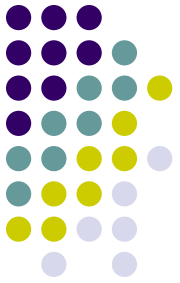
Symbol Table



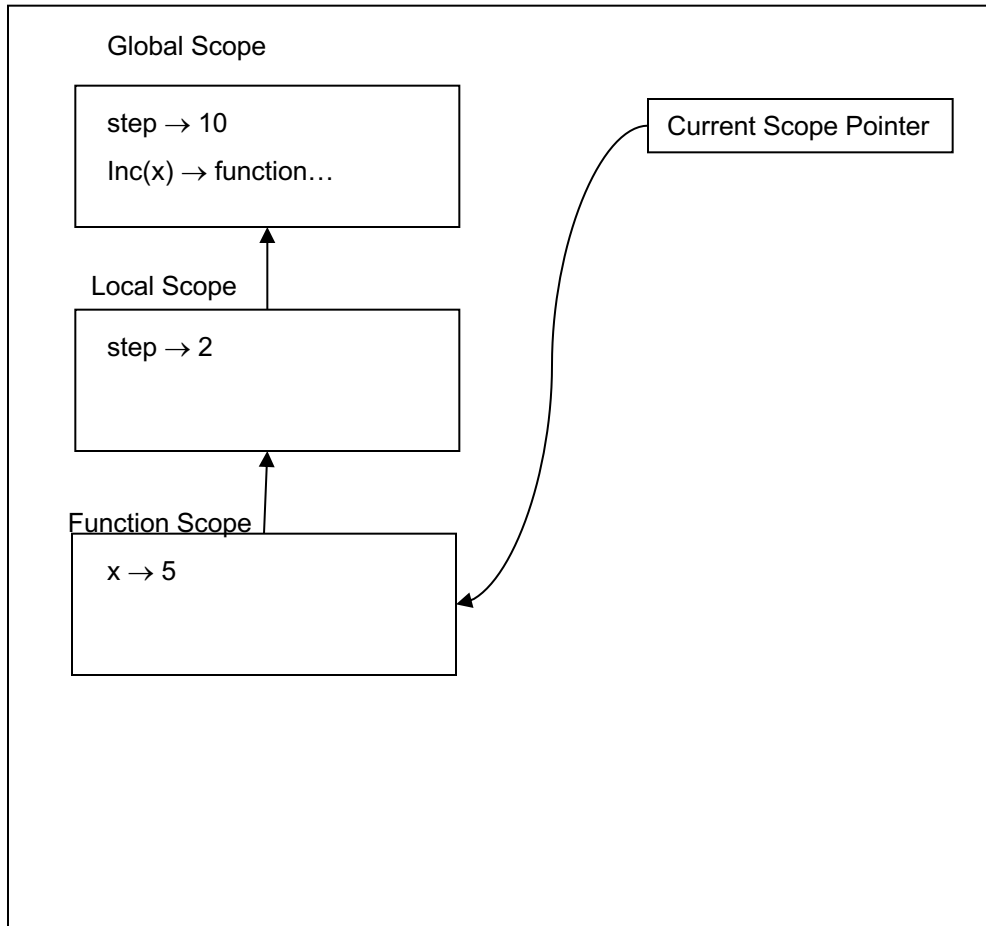
→

```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Dynamic Scoping



Symbol Table

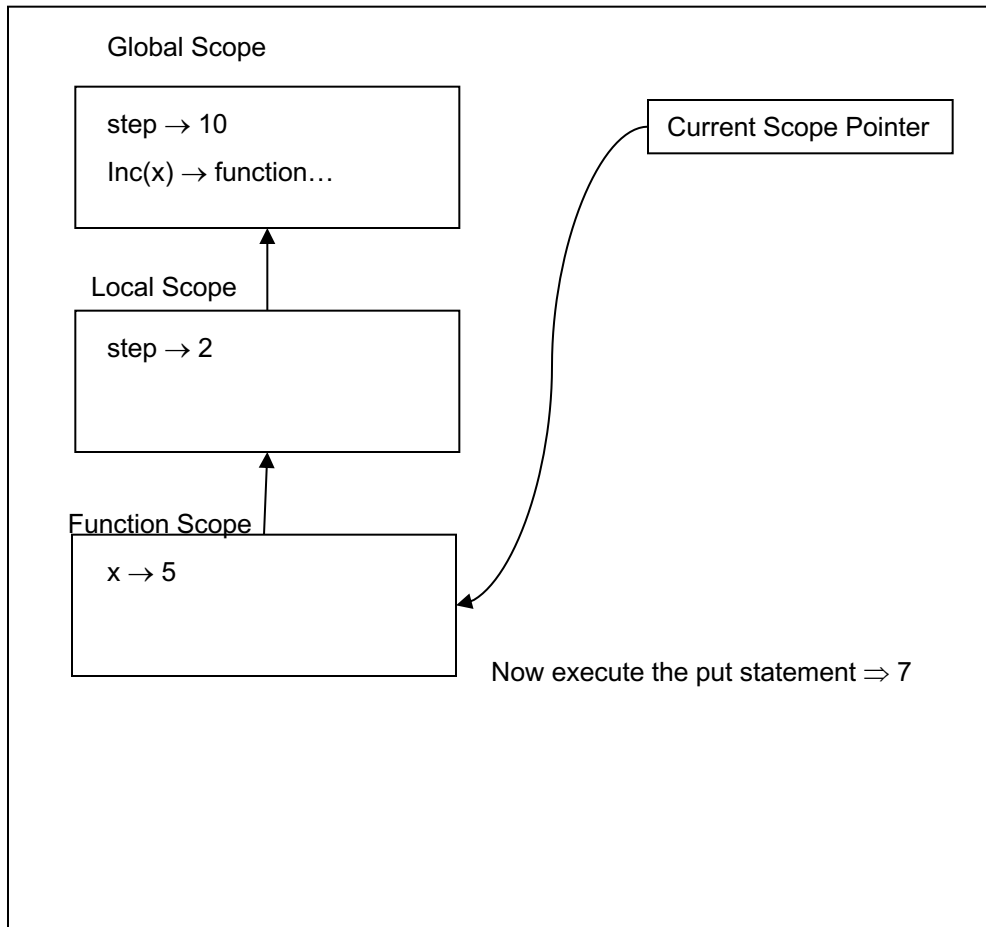


```
declare step = 10;  
declare inc(x) {  
    return x+step;  
}  
// start a local scope...  
{  
    declare step = 2;  
    put inc(5);  
}
```

# Dynamic Scoping



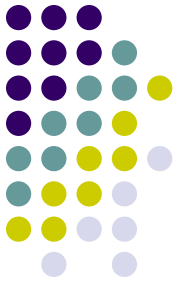
Symbol Table



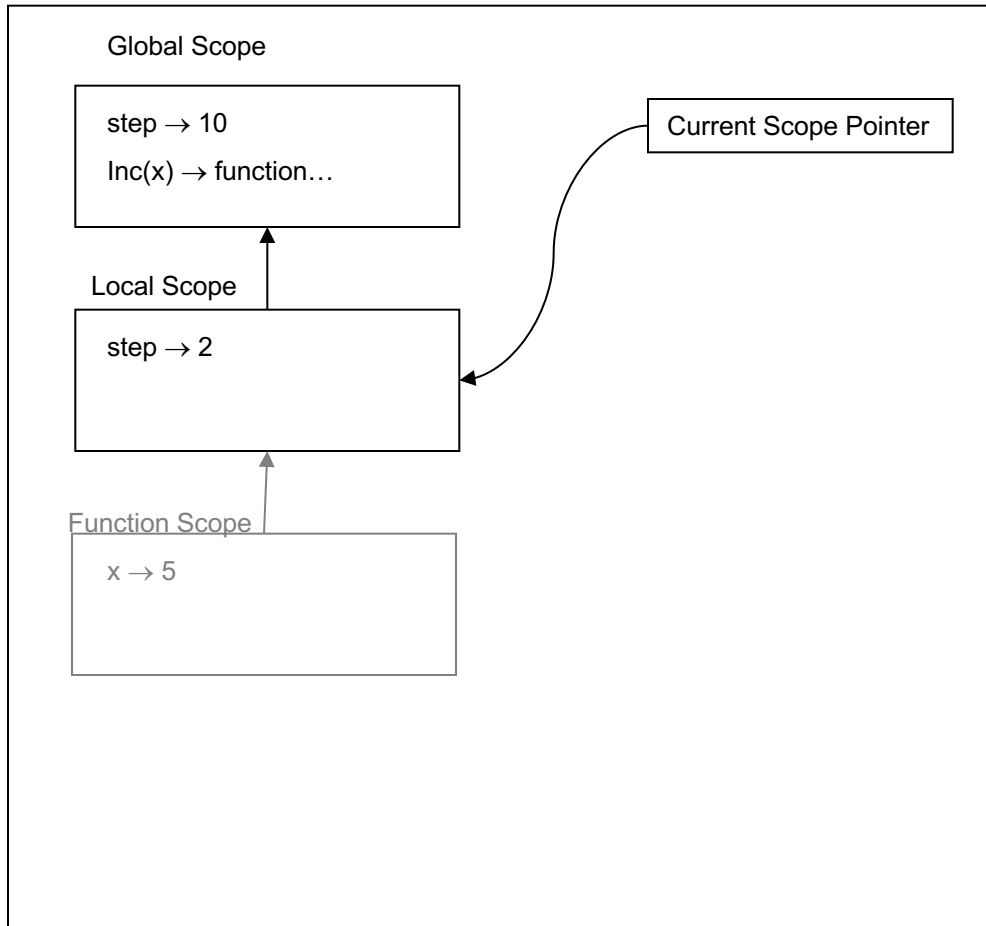
→

```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Dynamic Scoping



Symbol Table

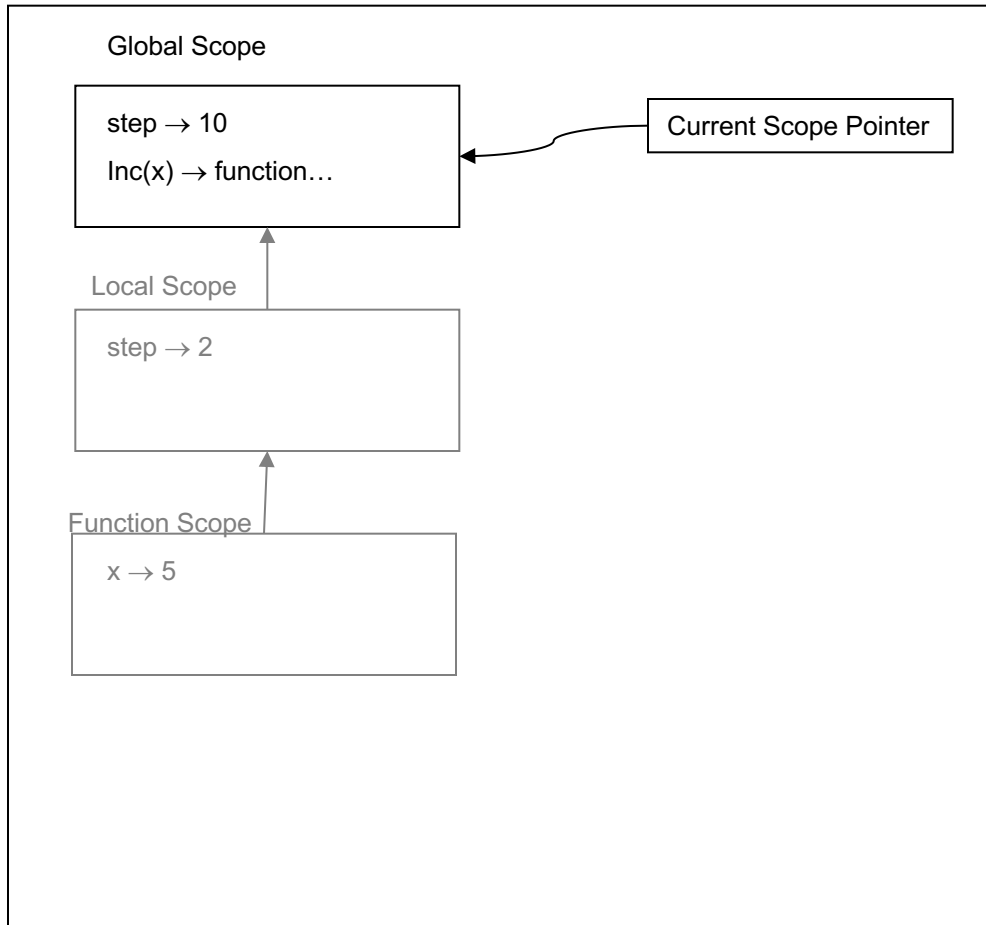


```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Dynamic Scoping



Symbol Table

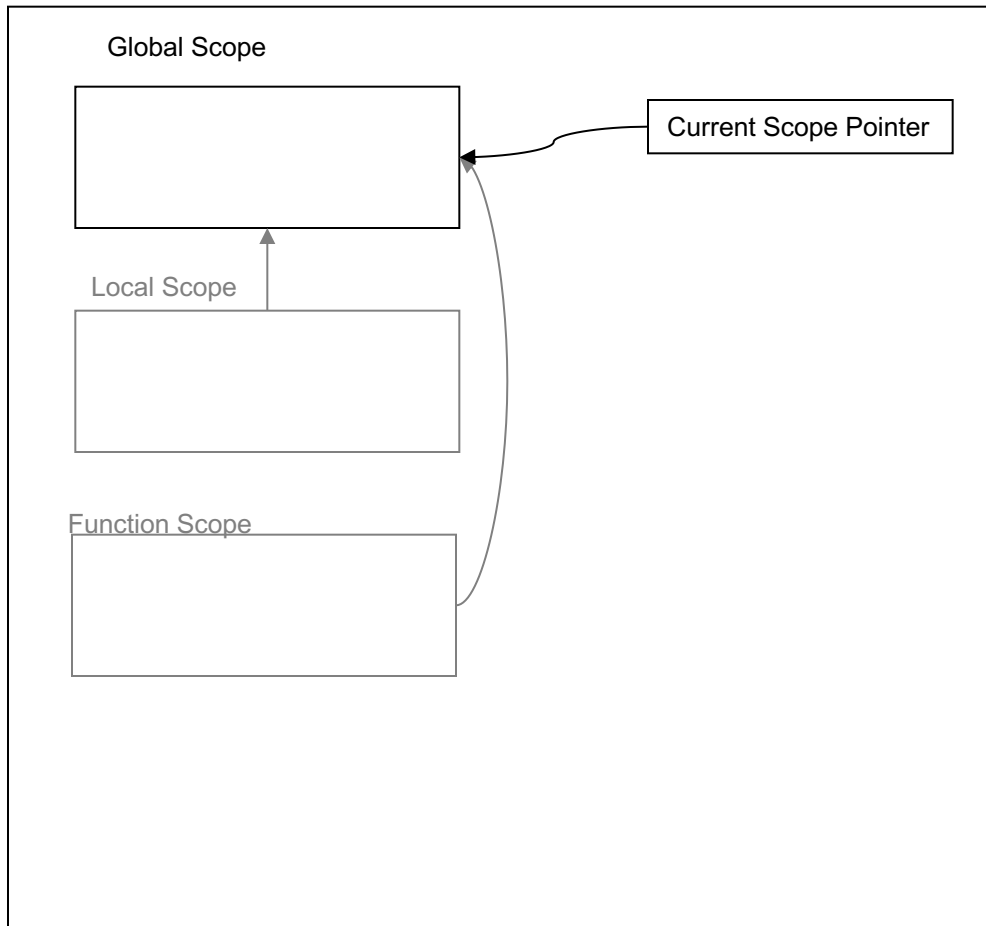


```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Static Scoping



Symbol Table

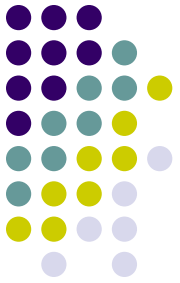


```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

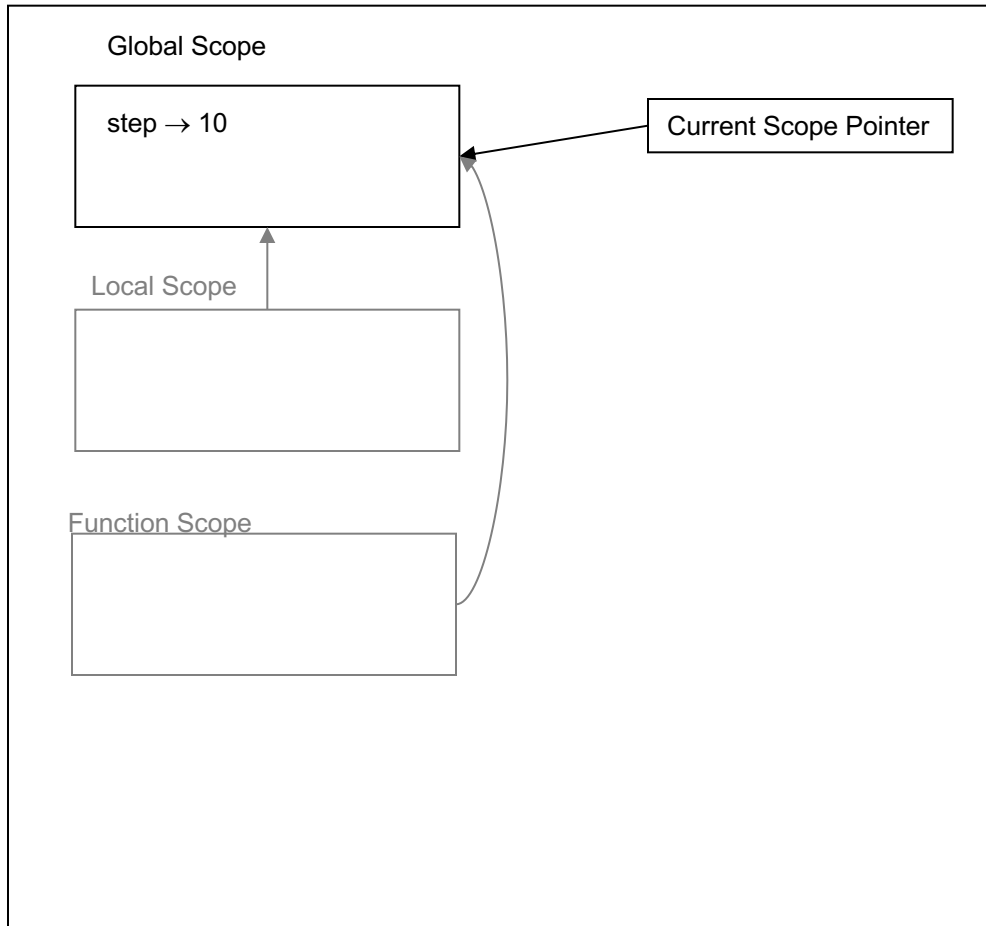
☞ In static scoping we push the function scope on scope stack but it remembers the scope it was declared in.

What is the expected output of the program under these assumptions?

# Static Scoping



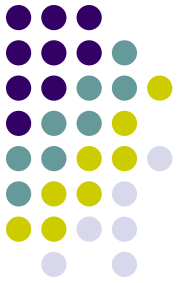
Symbol Table



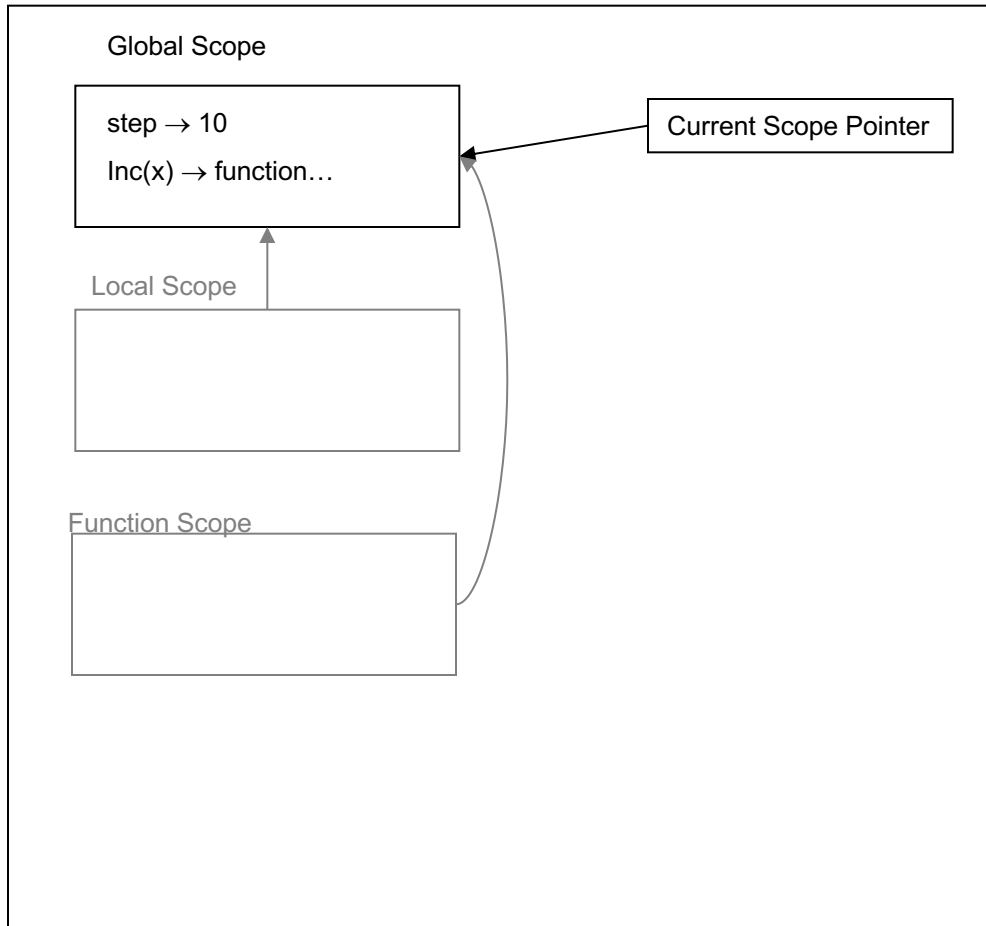
→ `declare step = 10;`  
`declare inc(x) {`  
    `return x+step;`  
`}`  
`// start a local scope...`  
`{`  
    `declare step = 2;`  
    `put inc(5);`  
`}`



# Static Scoping



Symbol Table

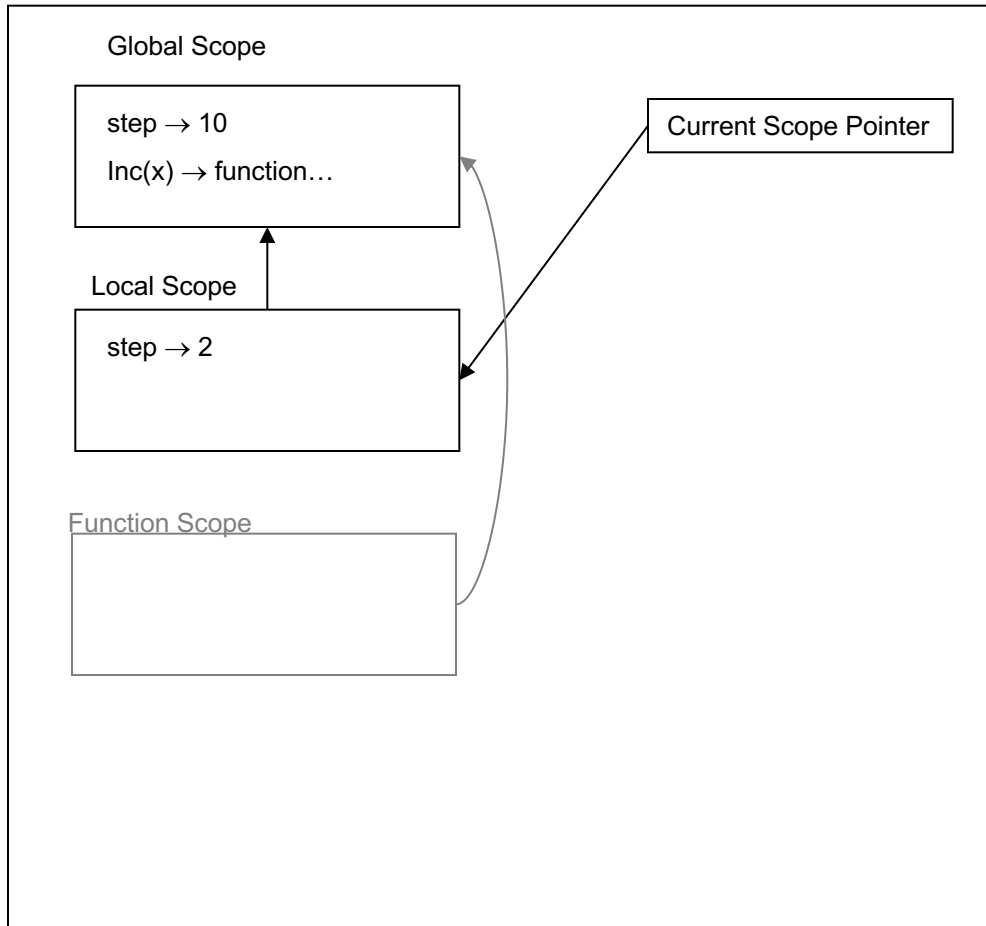


```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Static Scoping



Symbol Table



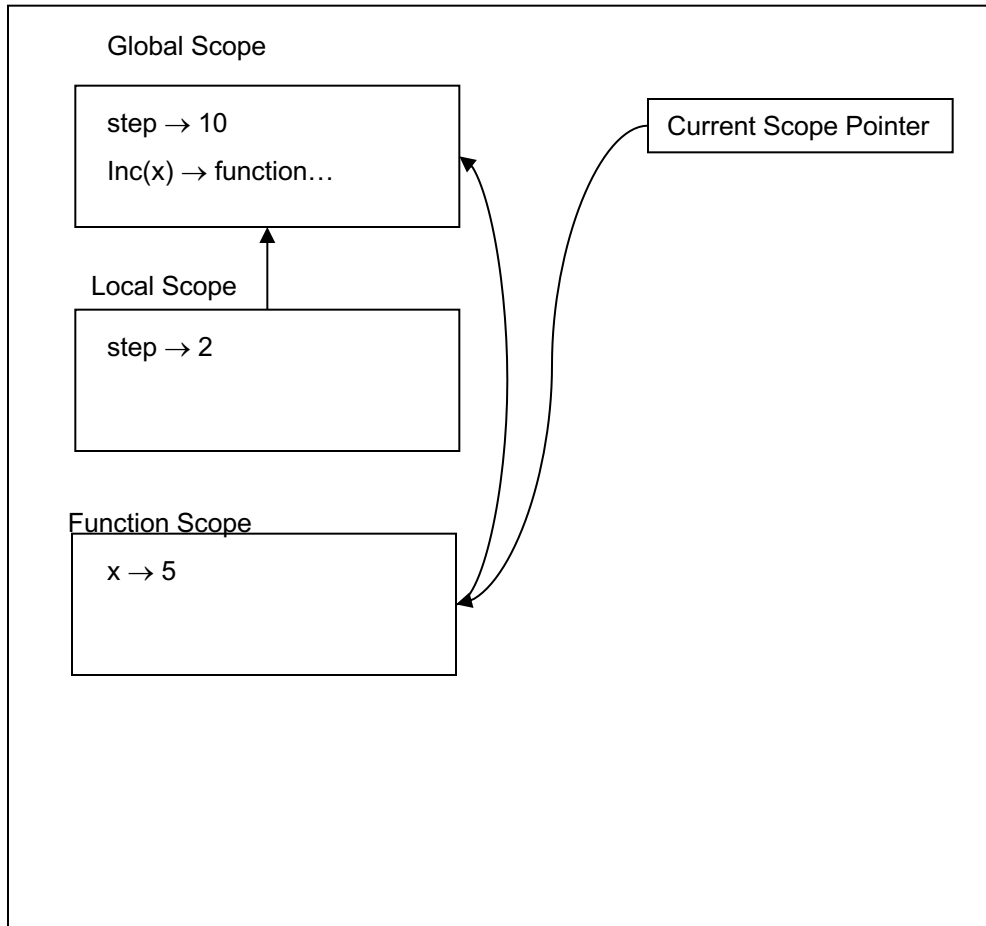
→

```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Static Scoping



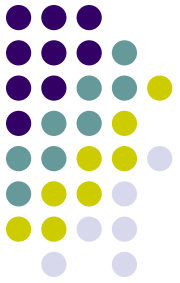
Symbol Table



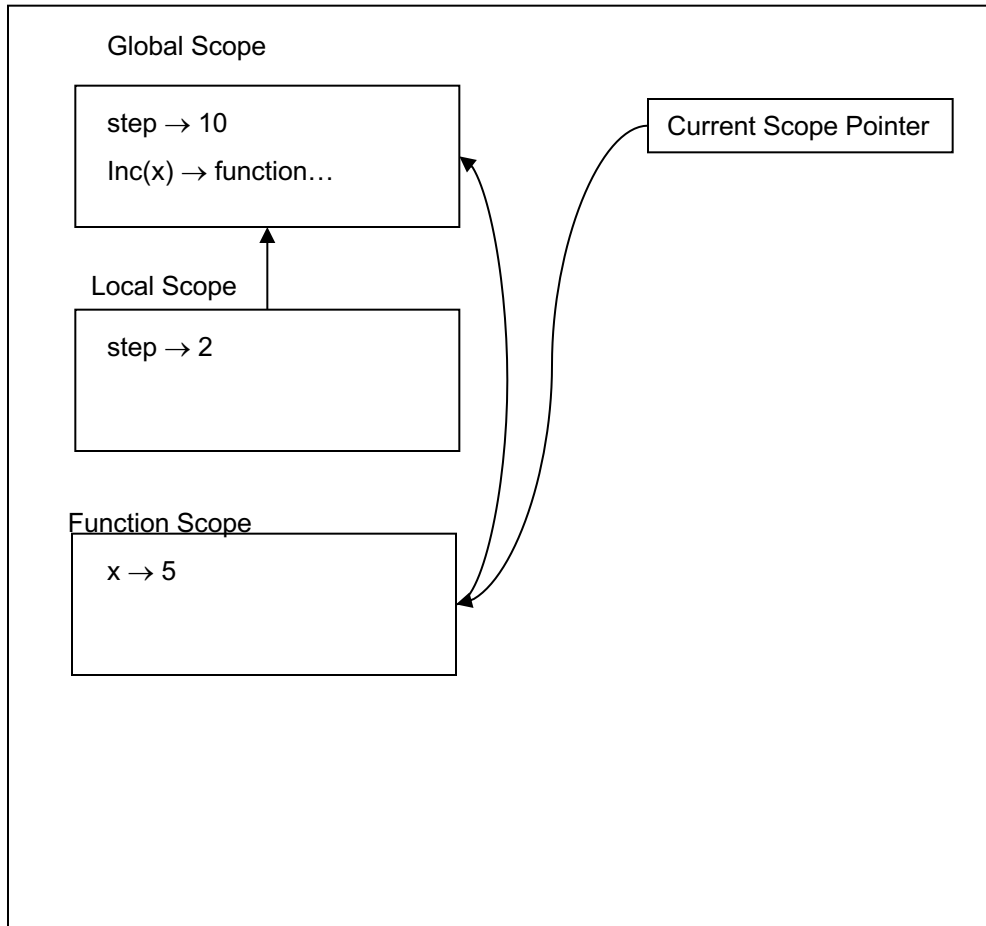
→

```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Static Scoping



Symbol Table



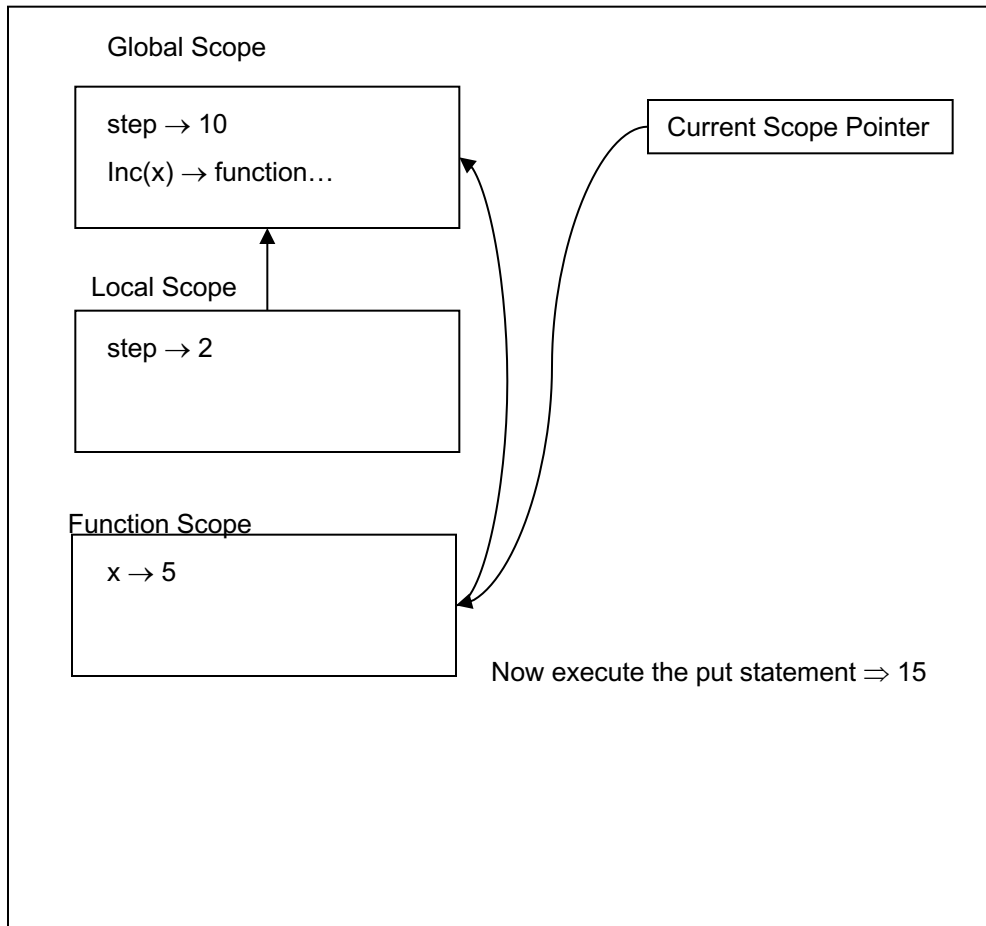
→


```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Static Scoping

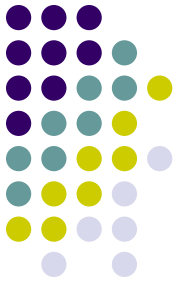


Symbol Table

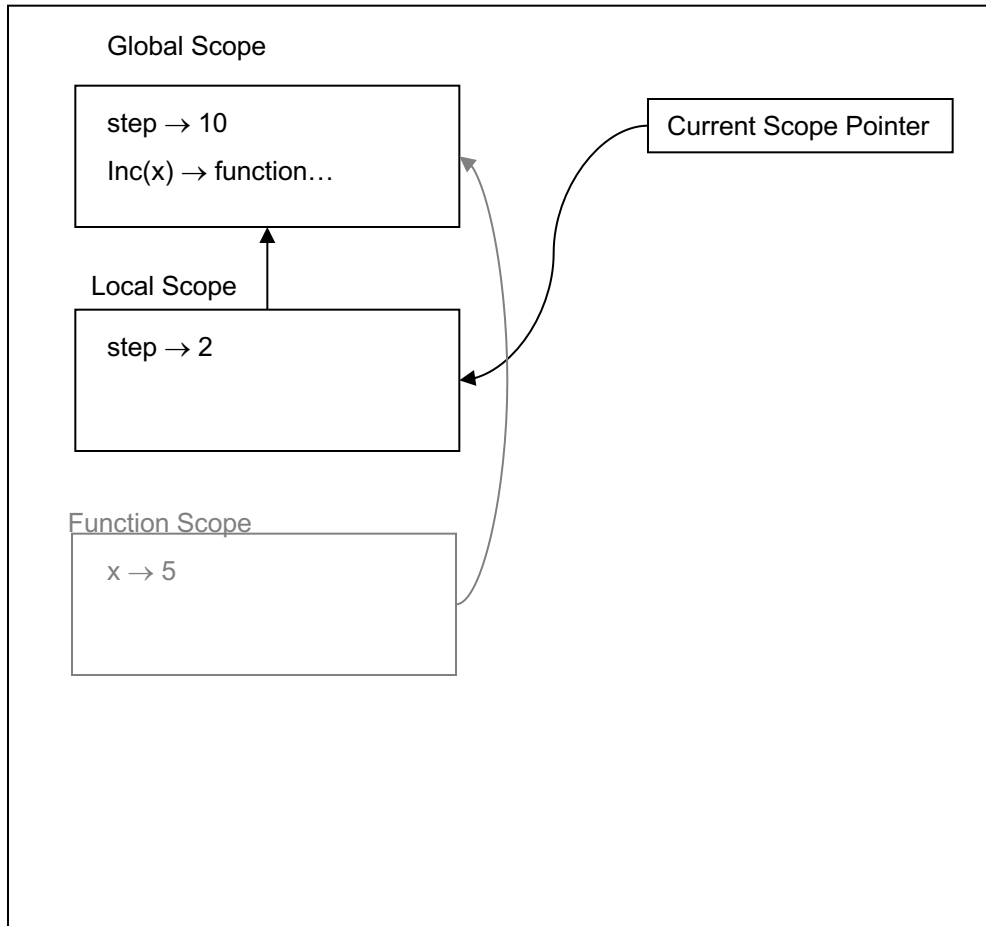


 `declare step = 10;  
declare inc(x) {  
 return x+step;  
}  
// start a local scope...  
{  
 declare step = 2;  
 put inc(5);  
}`

# Static Scoping



Symbol Table

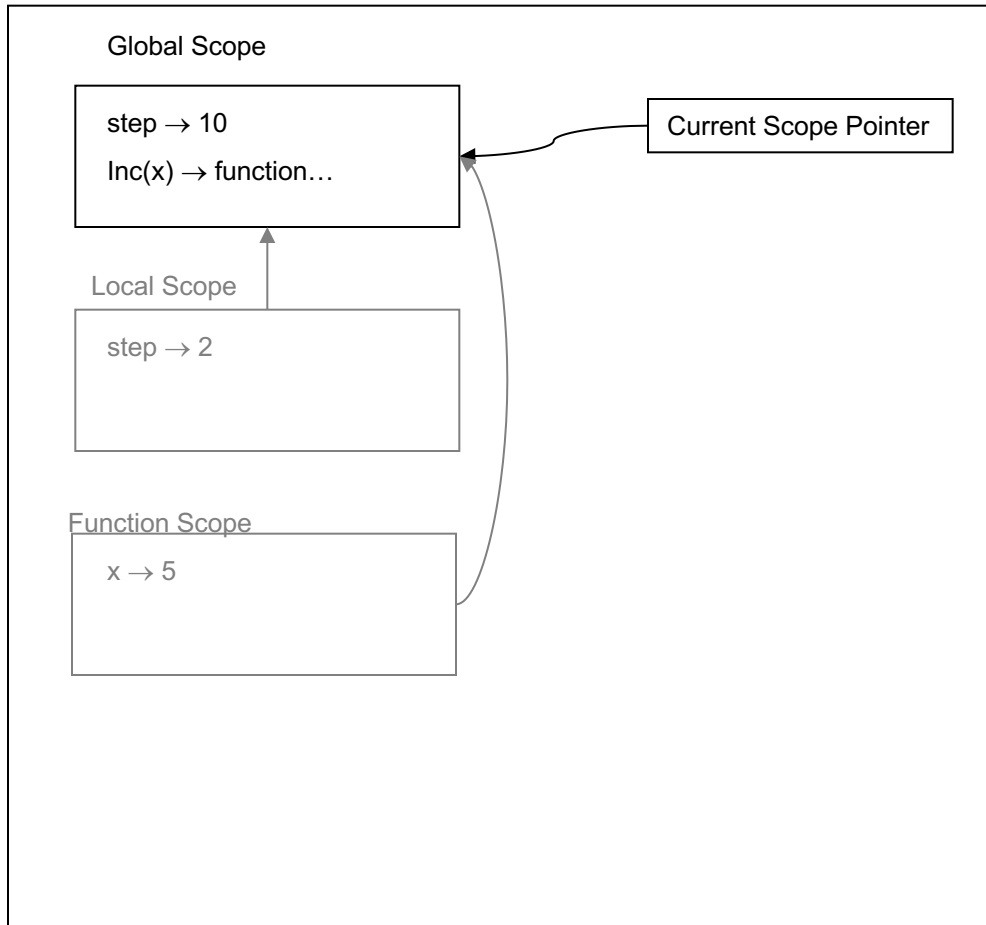


```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

# Static Scoping



Symbol Table



```
declare step = 10;
declare inc(x) {
    return x+step;
}
// start a local scope...
{
    declare step = 2;
    put inc(5);
}
```

☞ It is clear that what we want is static scoping.

# Assignment

- Final project proposal

