Higher Order Programming

- Let's play a syntactic trick…
  - We already decided that function names really have no influence on the actual function definition, i.e., the behavior of the function does not depend on its name. That is 'declare inc(x) return x+1' is the same function as 'declare icecream(x) return x+1'.
  - We took advantage of this in our implementation by using the name as a handle to retrieve the actual function value from the symbol table during interpretation of a program.
  - The idea then is to make this separation of name and behavior visible at the syntactic level

Higher Order Programming

- That is the function declaration

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

  becomes

  ```
  declare inc = function (x) return x+1;
  ```

- Notice, that all we have done is to make the separation of function name and function behavior explicit at the syntactic level.

Higher Order Programming

- However, this simple trick has huge ramifications in terms of expressive power of the programming language.
- First consider that the function declaration now looks like any variable declaration statement:

```
declare inc = function (x) return x+1;
```

Init value

Variable name

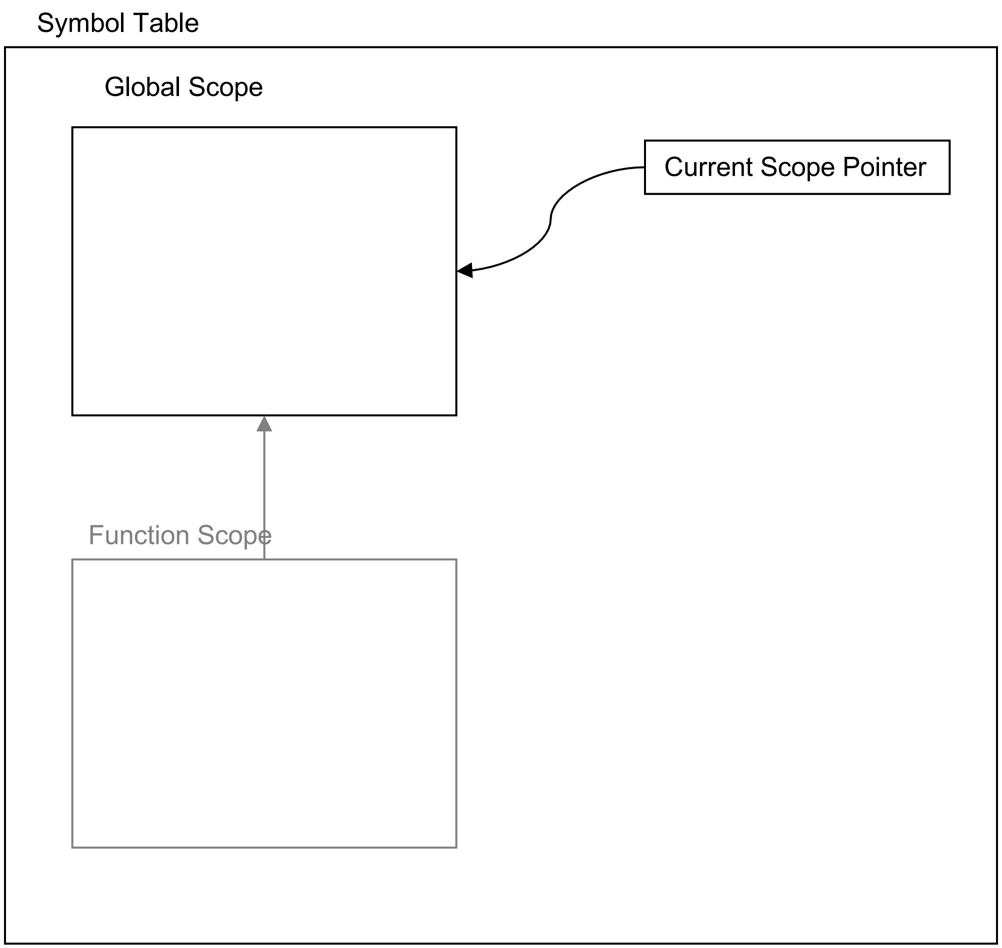
☞ The function behavior is the value assigned to the variable ‘inc’

Higher Order Programming

- Now, consider the ramifications…if the function behavior is just a value then there is nothing to prevent us from assigning that value to another variable.
- But that means that the new variable name is now also a handle to access the function behavior

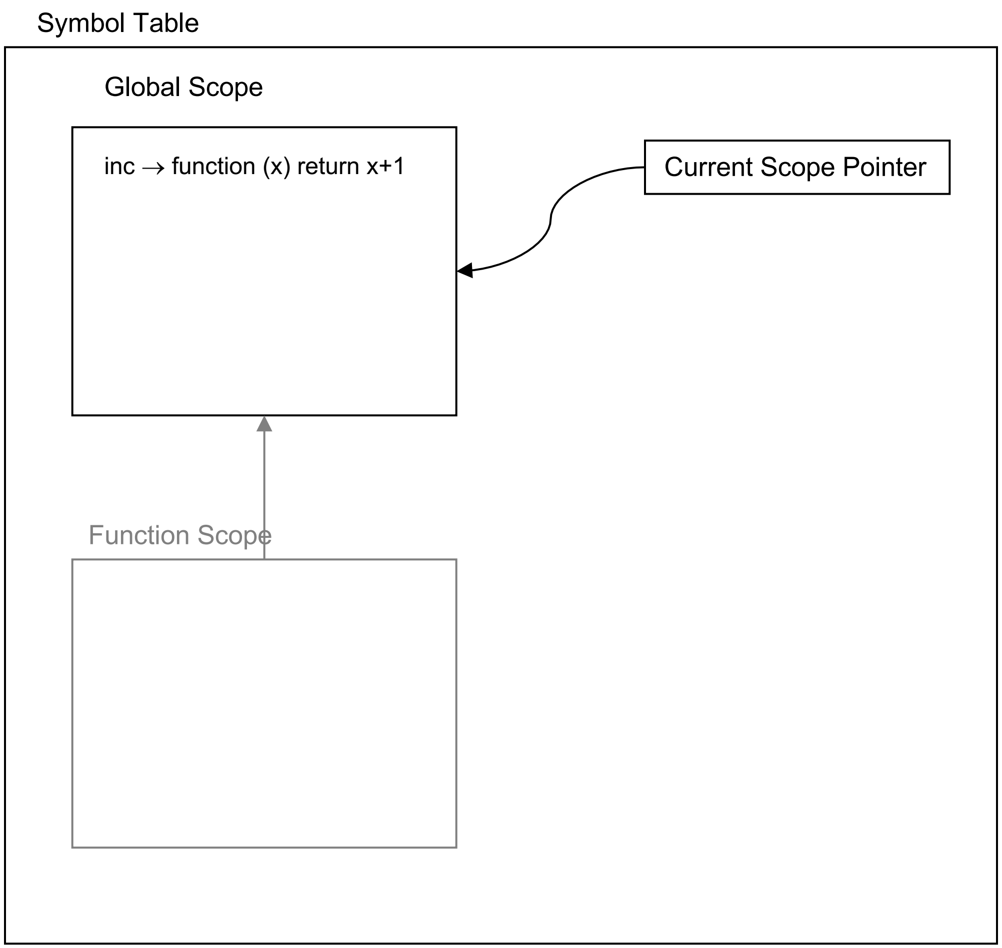
```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

Interpreting Functions



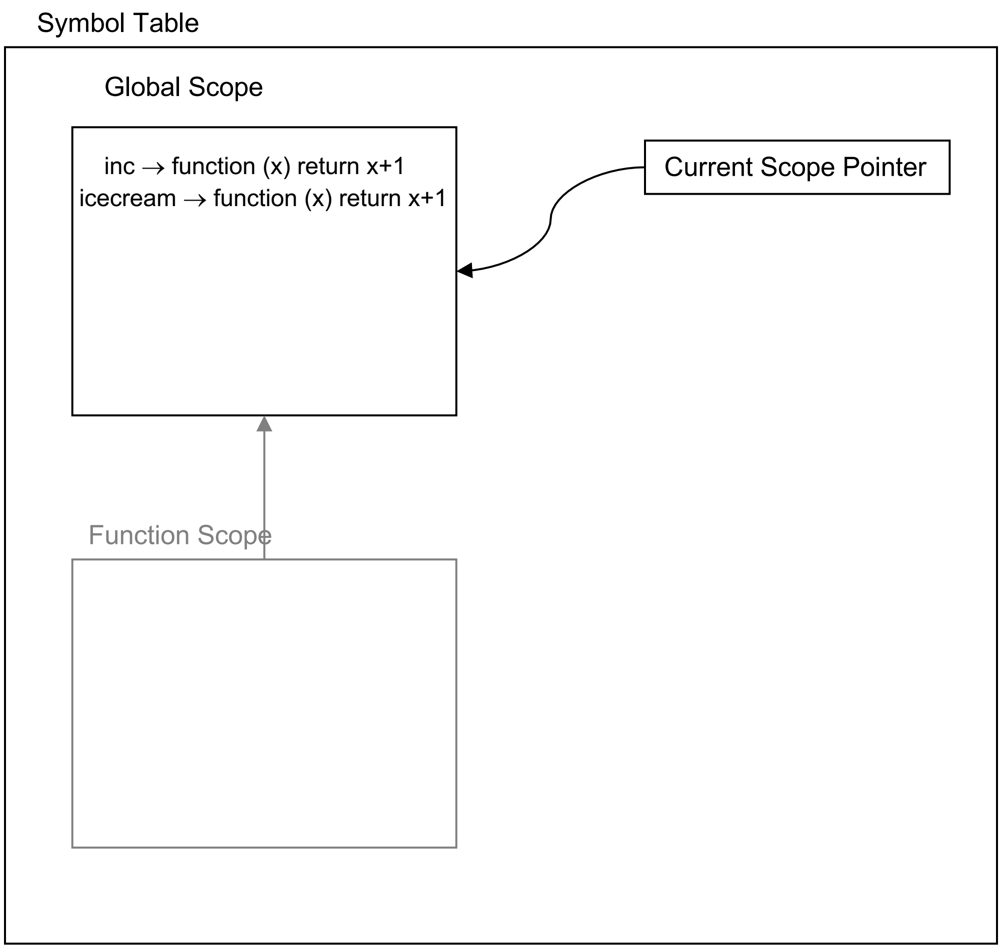

```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

Interpreting Functions



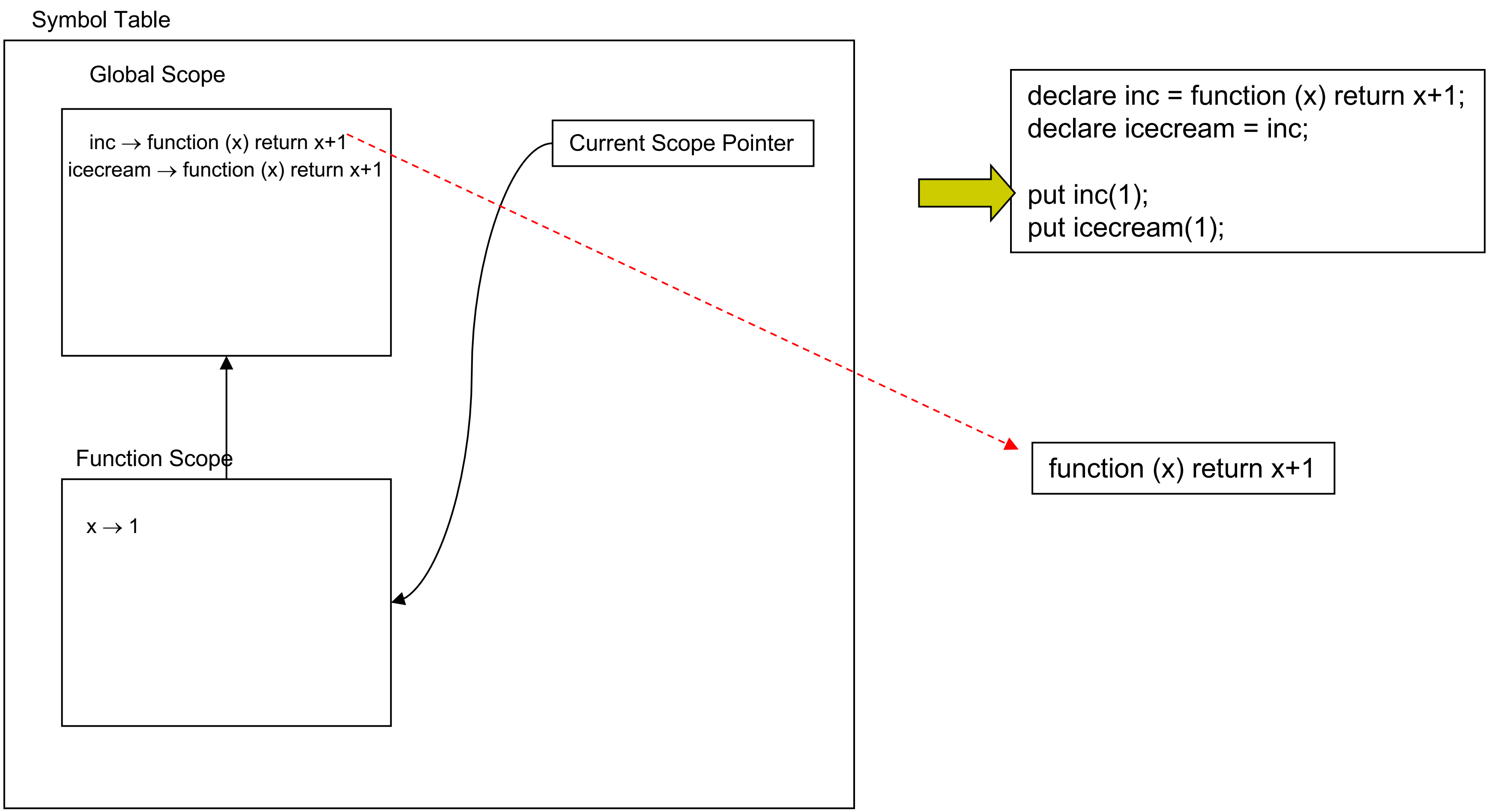

```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

Interpreting Functions

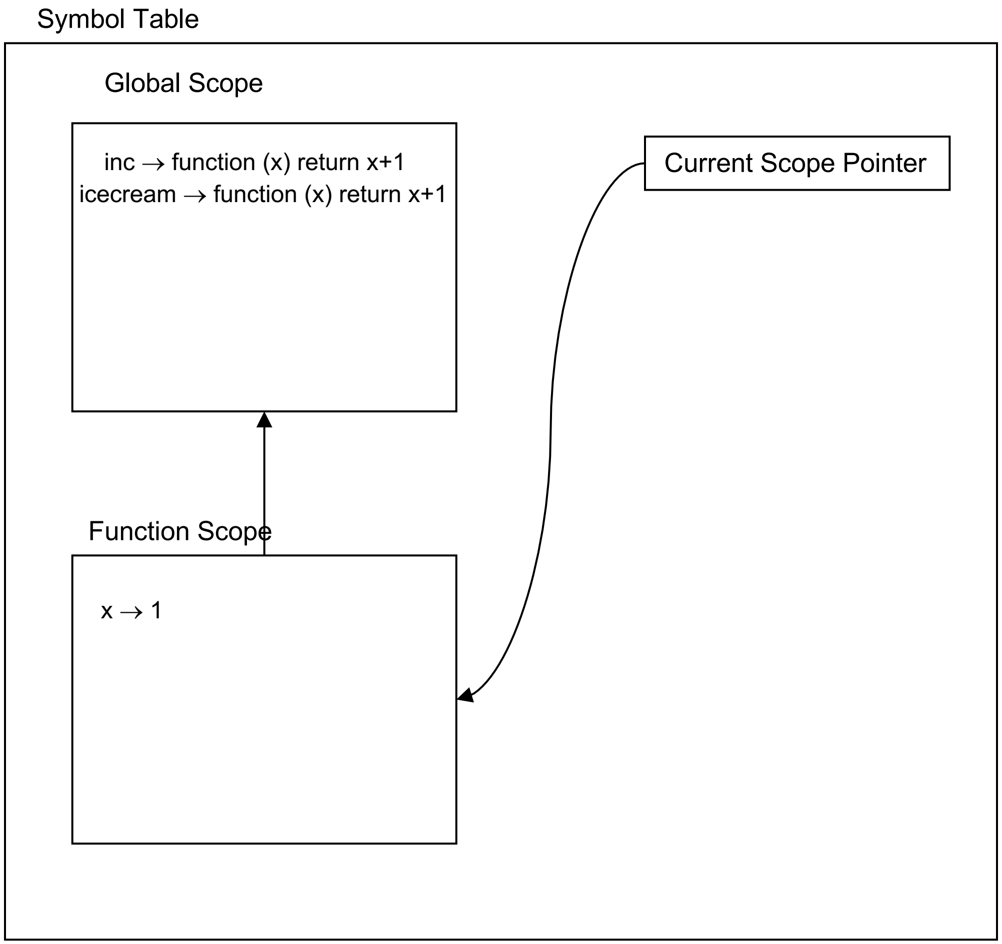



```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

Interpreting Functions



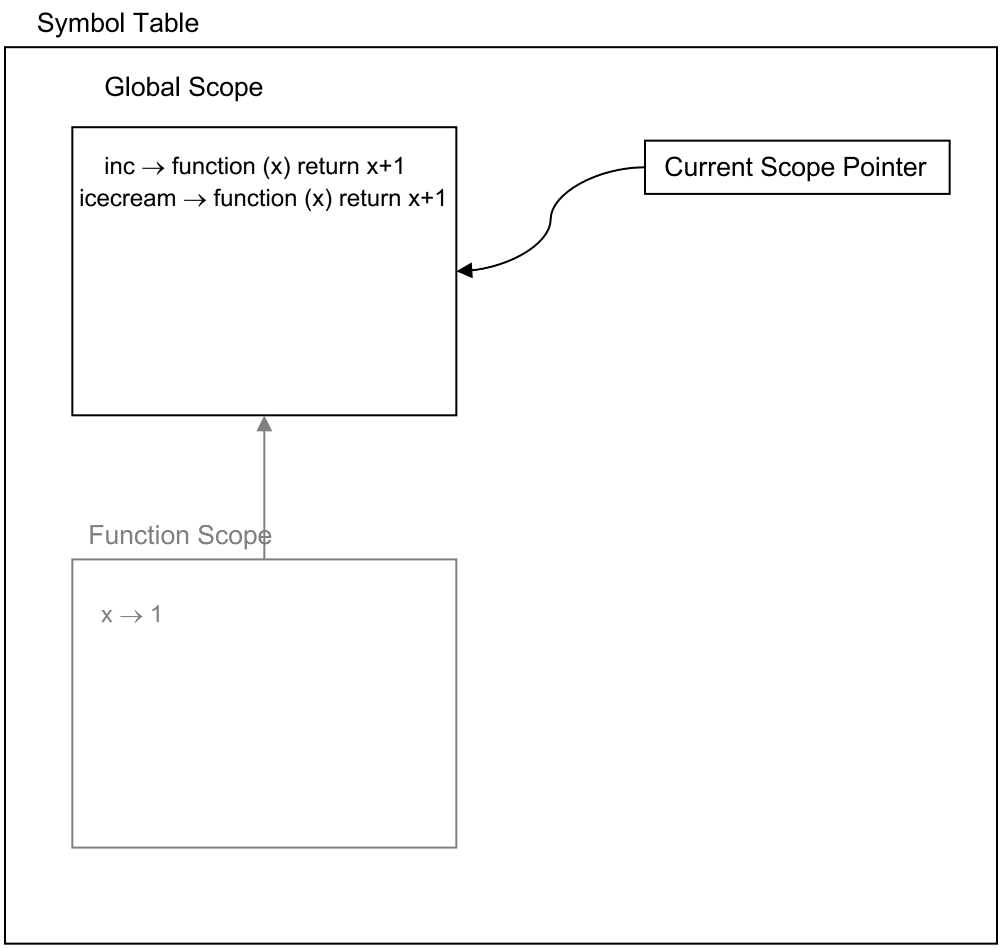
Interpreting Functions




```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

Interpreting Functions

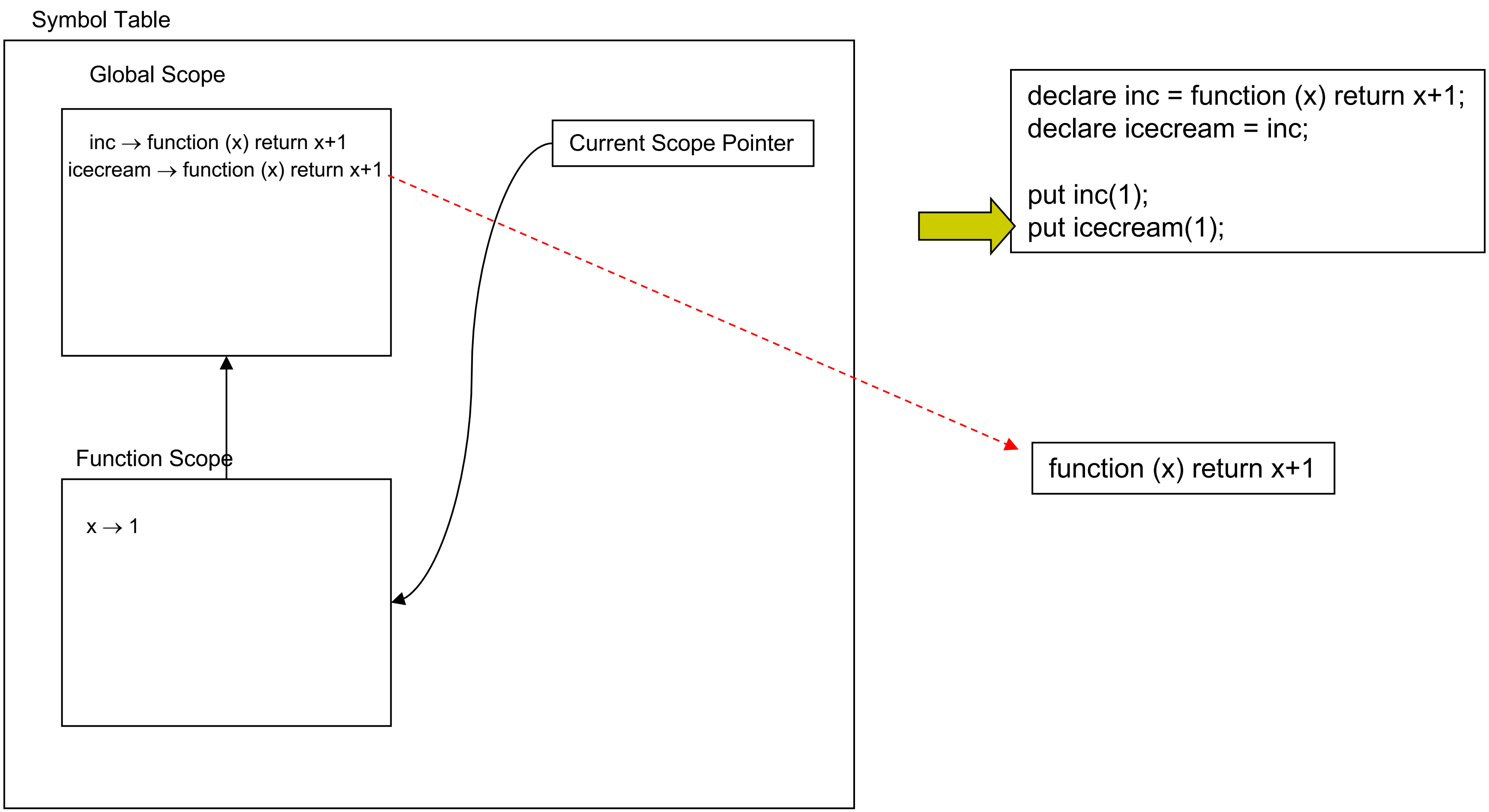
2




```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

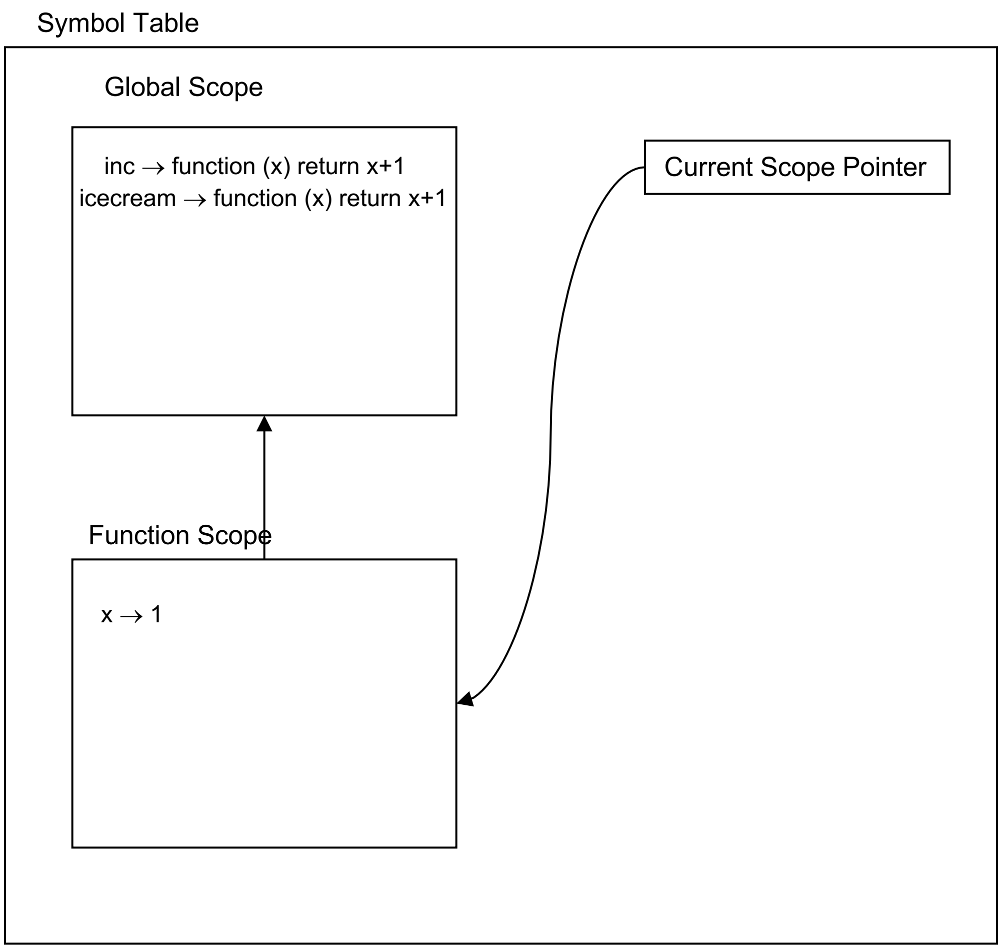
Interpreting Functions

2



Interpreting Functions

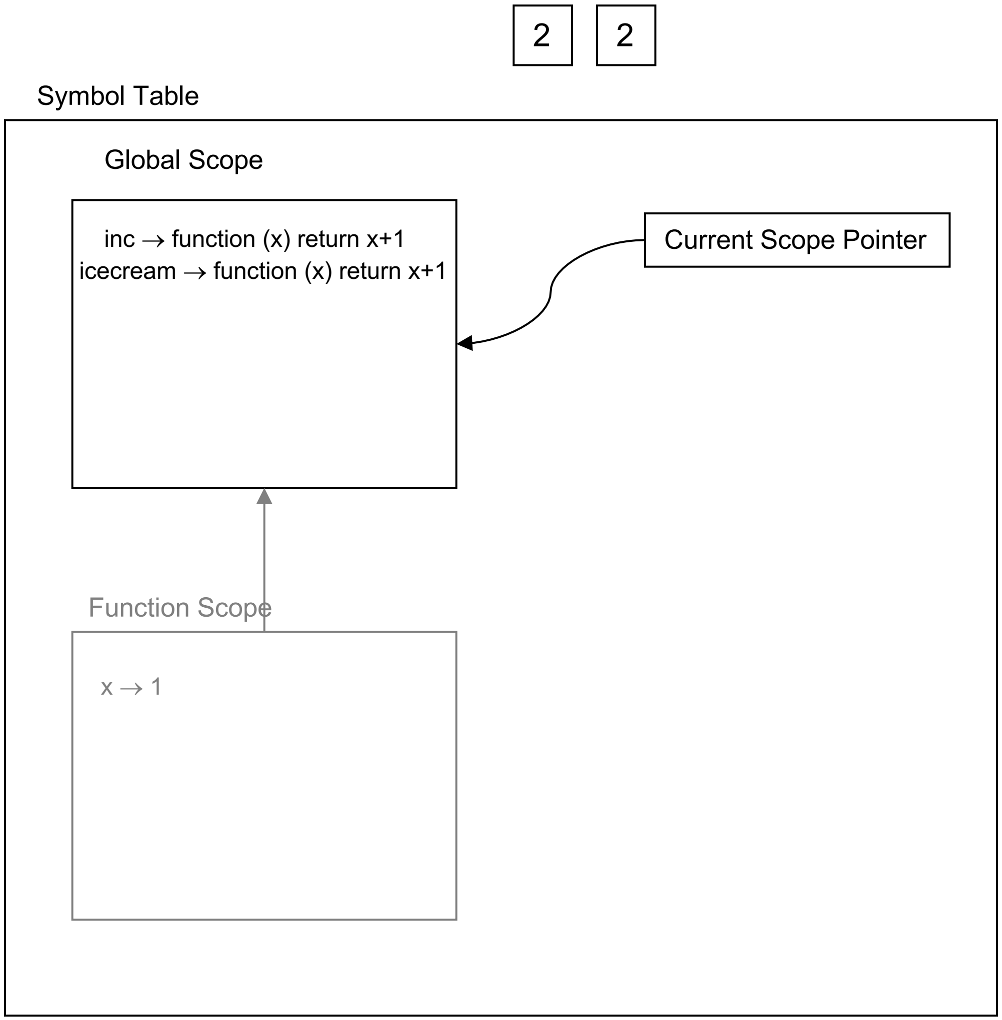
2




```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

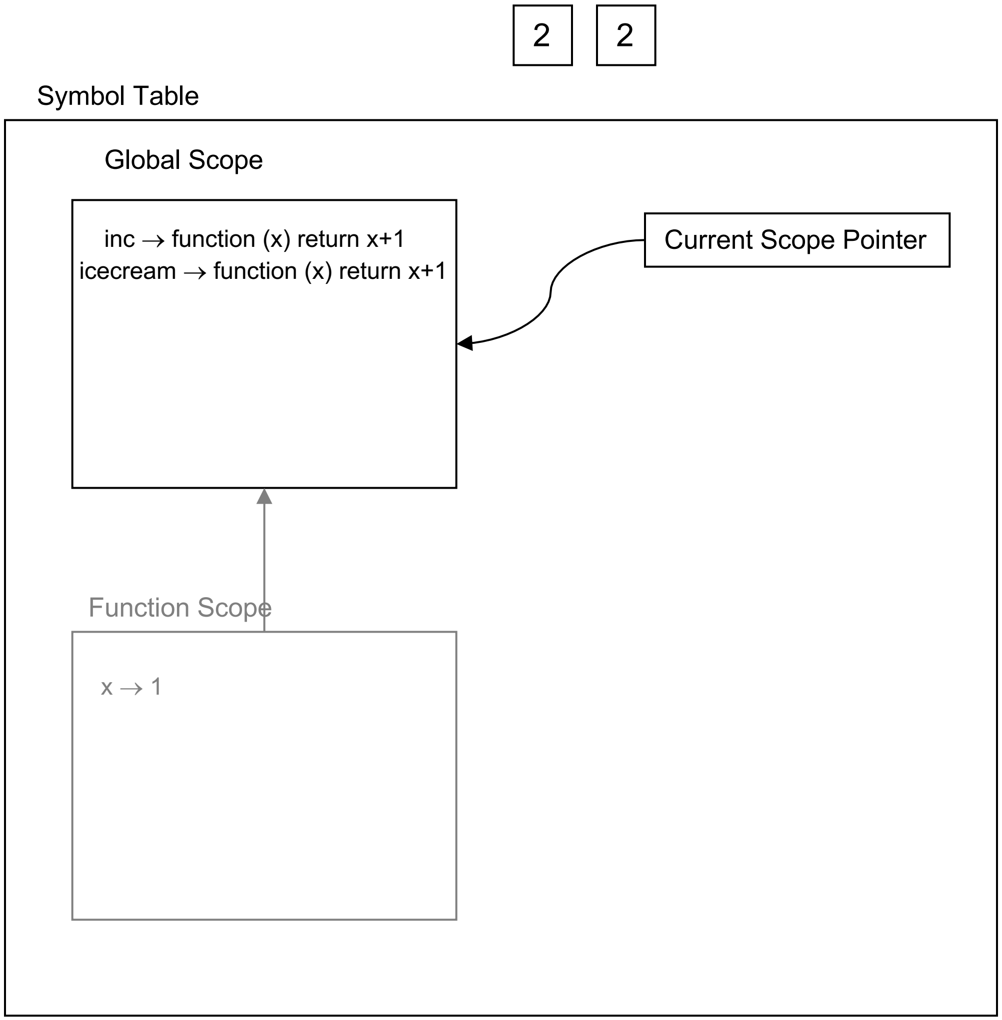
function (x) return x+1

Interpreting Functions




```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

Interpreting Functions



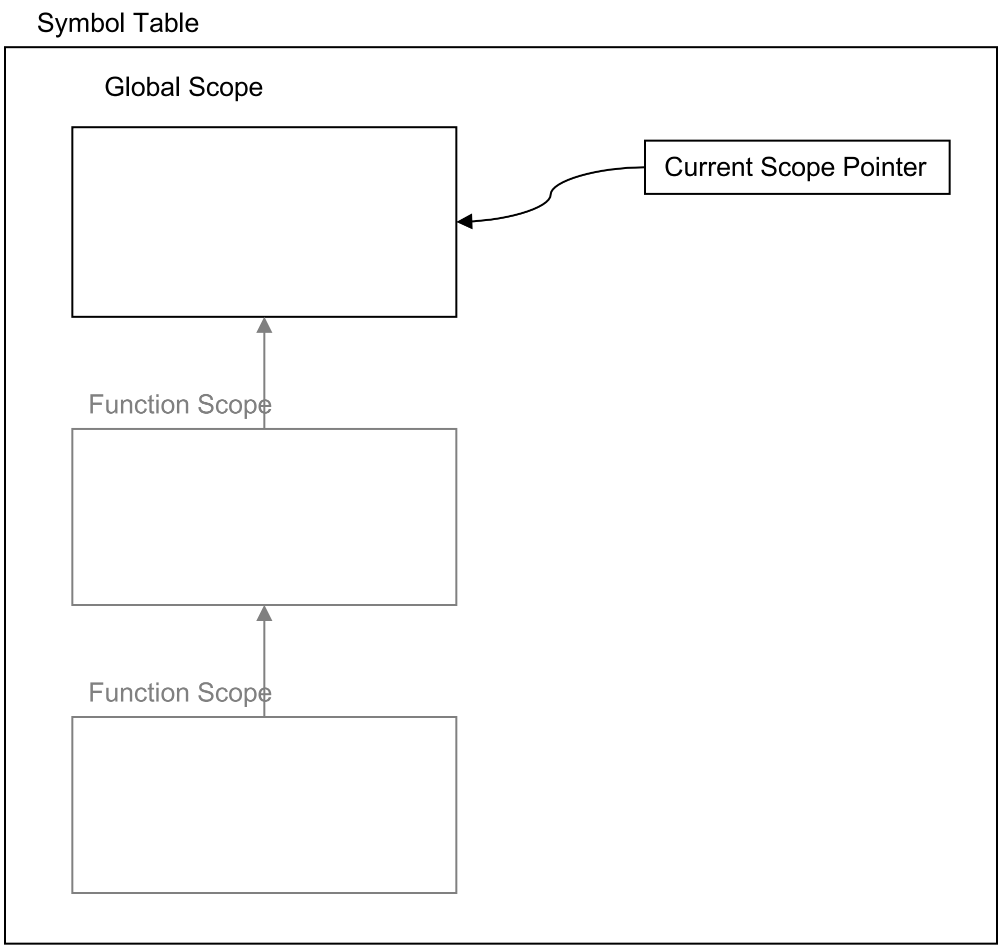

```
declare inc = function (x) return x+1;
declare icecream = inc;

put inc(1);
put icecream(1);
```

Higher Order Programming

- Now consider, formal function parameters are also variables, that means we can assign function values to the formal parameters of functions
- That is, we can pass functions to functions!

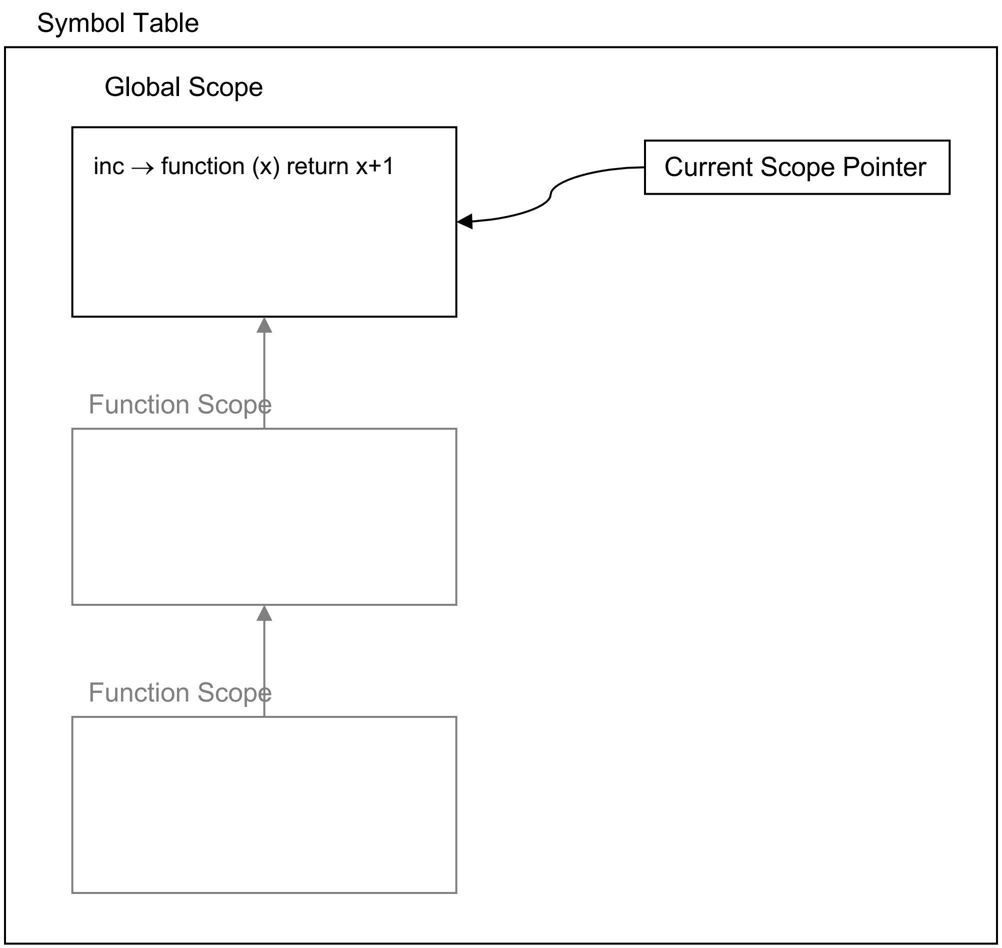
```
declare inc = function (x) return x+1;
declare apply = function(f,x) return f(x);

put apply(inc,1);
```

Interpreting Functions



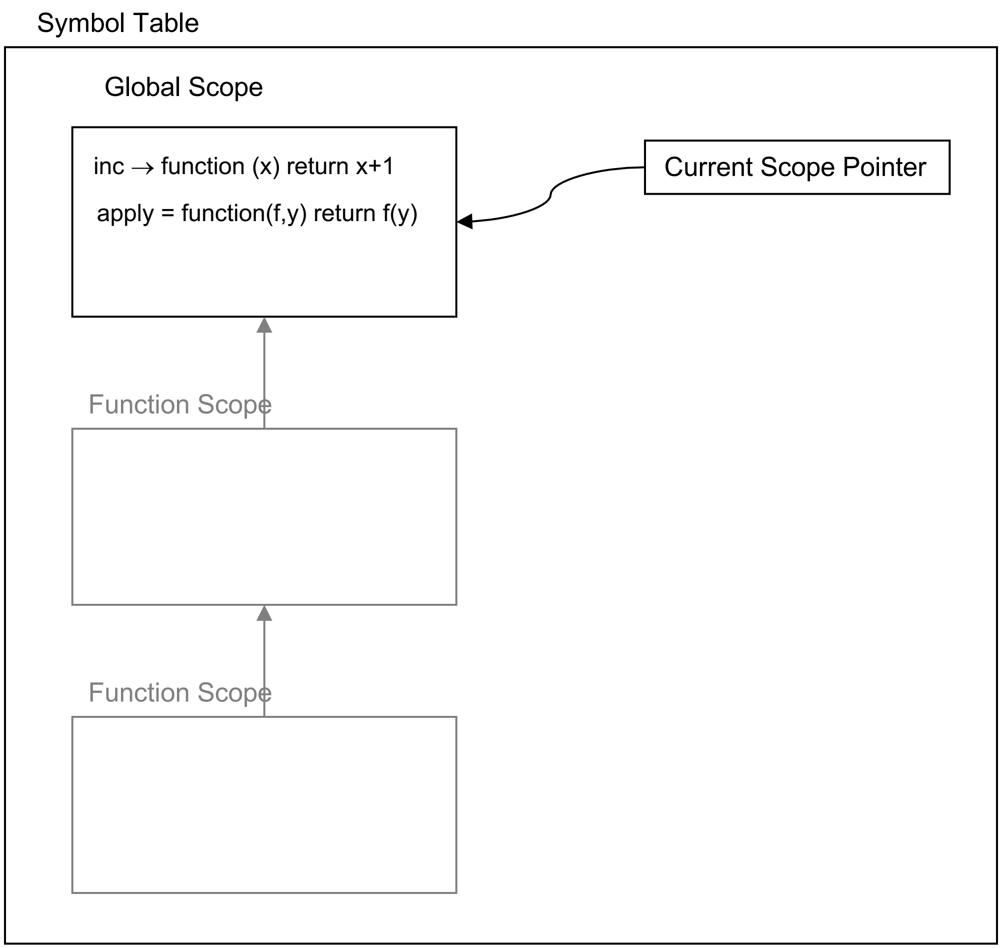

```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

Interpreting Functions



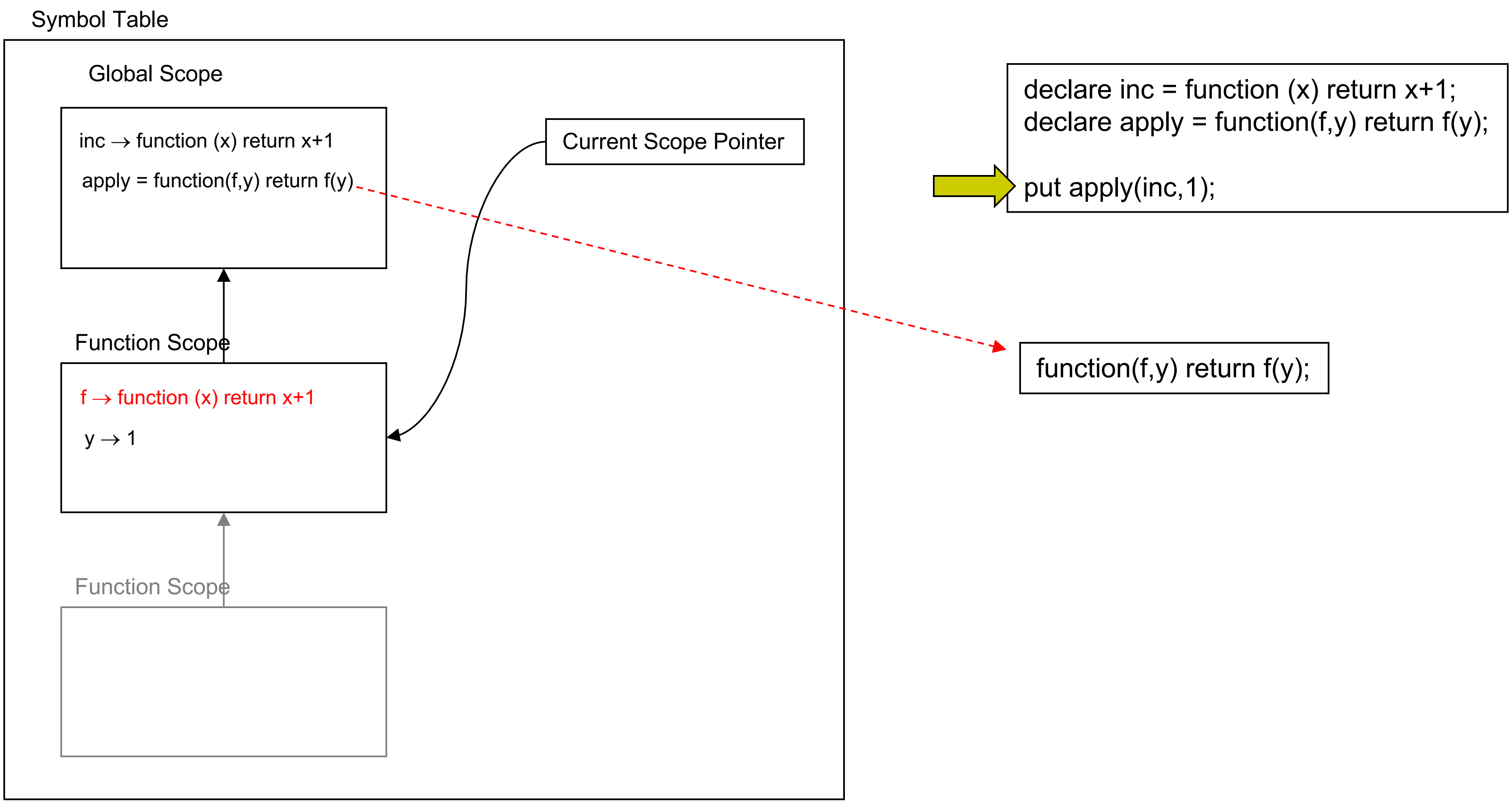

```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

Interpreting Functions

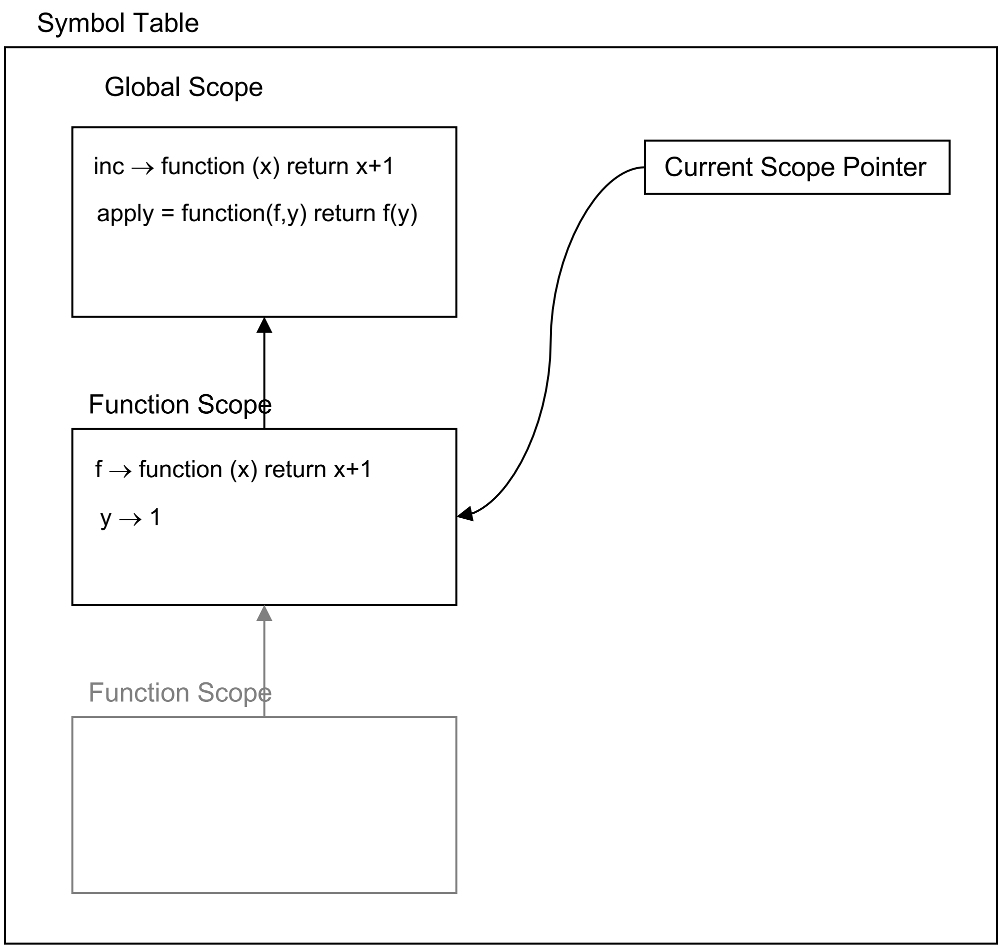



```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

Interpreting Functions



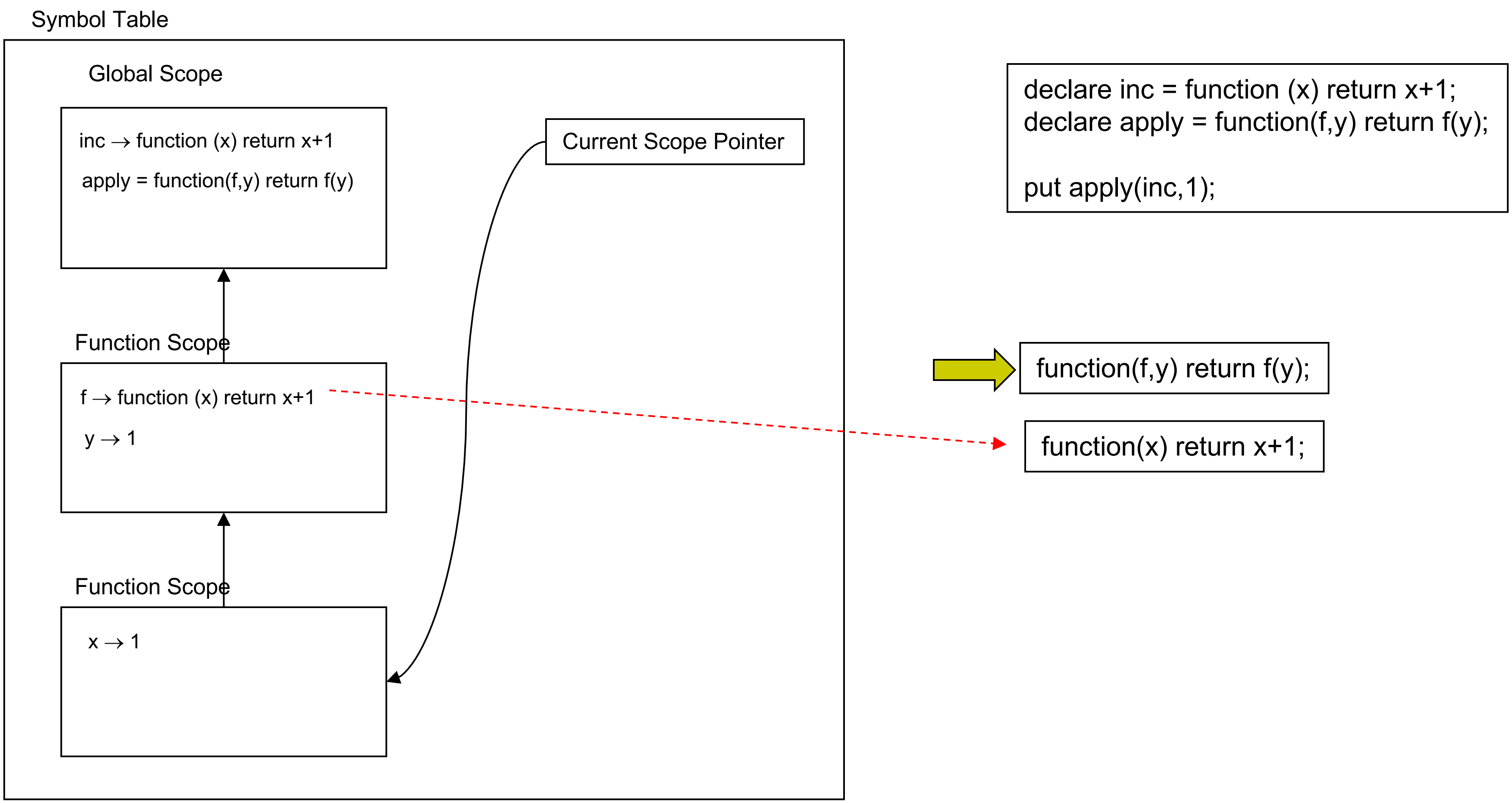
Interpreting Functions



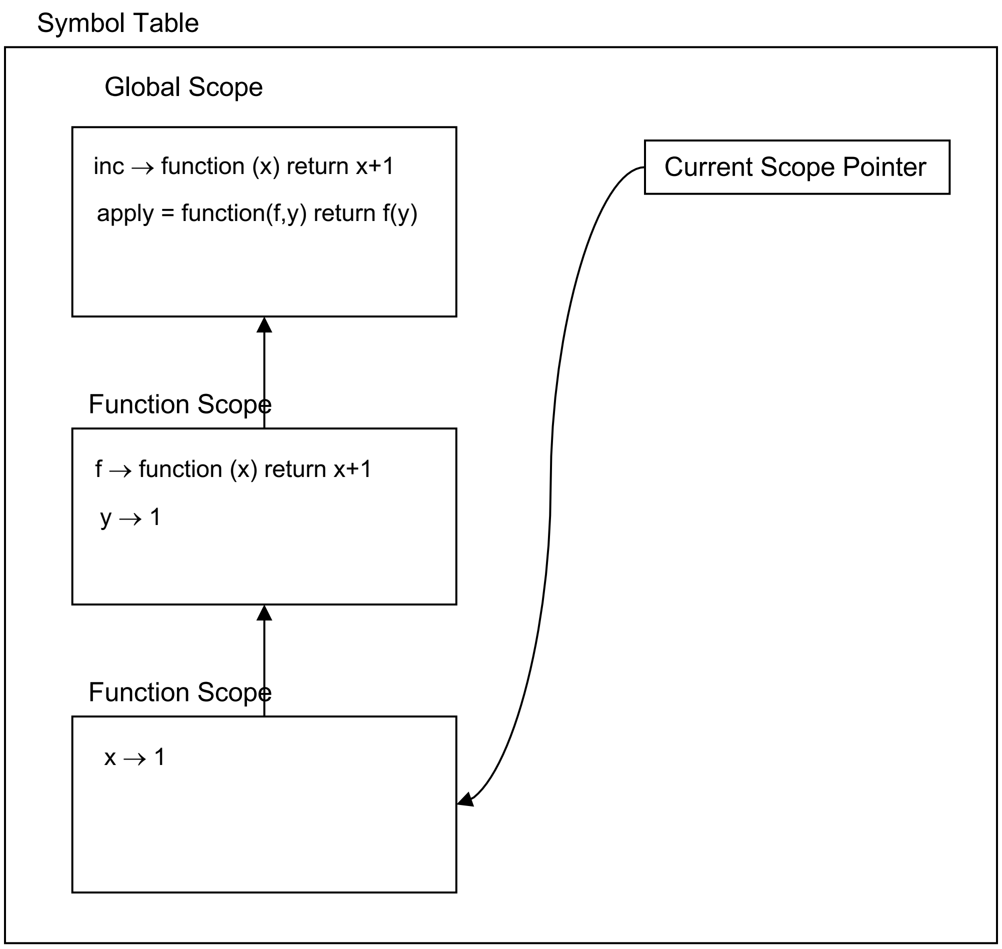

```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

```
function(f,y) return f(y);
```

Interpreting Functions



Interpreting Functions

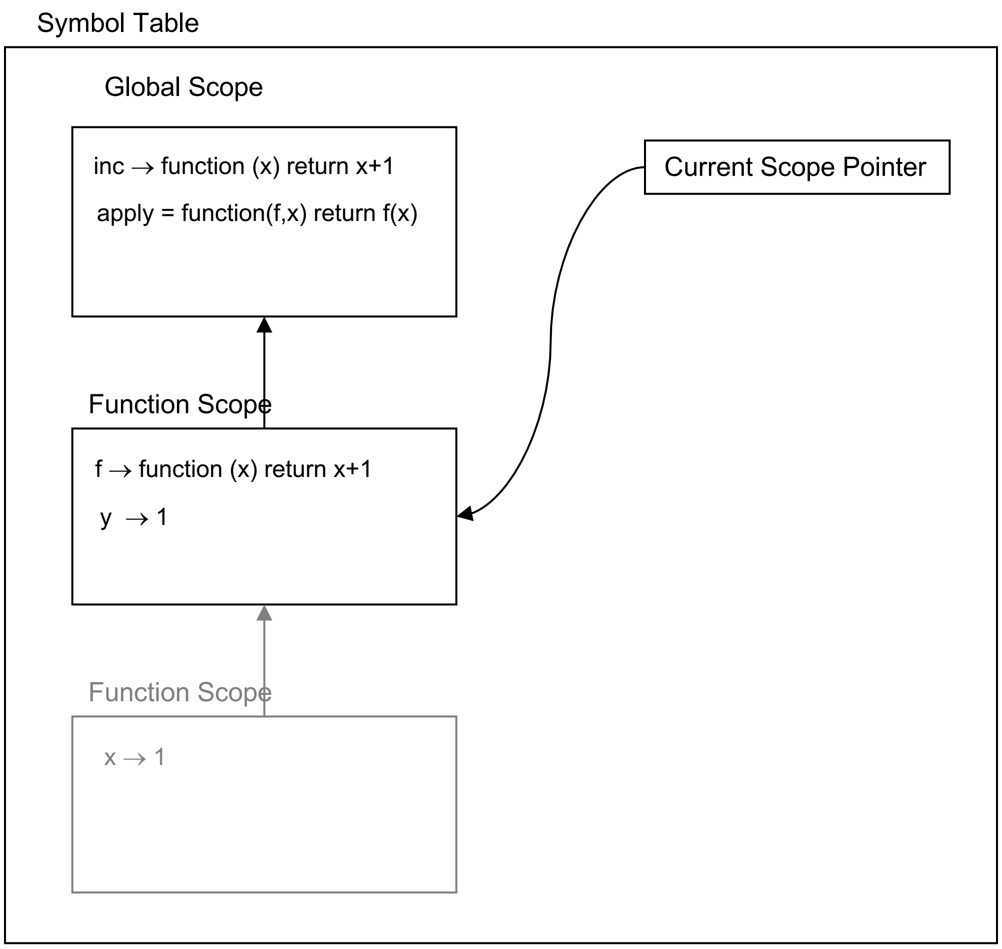



```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

```
function(f,y) return f(y);
```

```
function(x) return x+1;
```

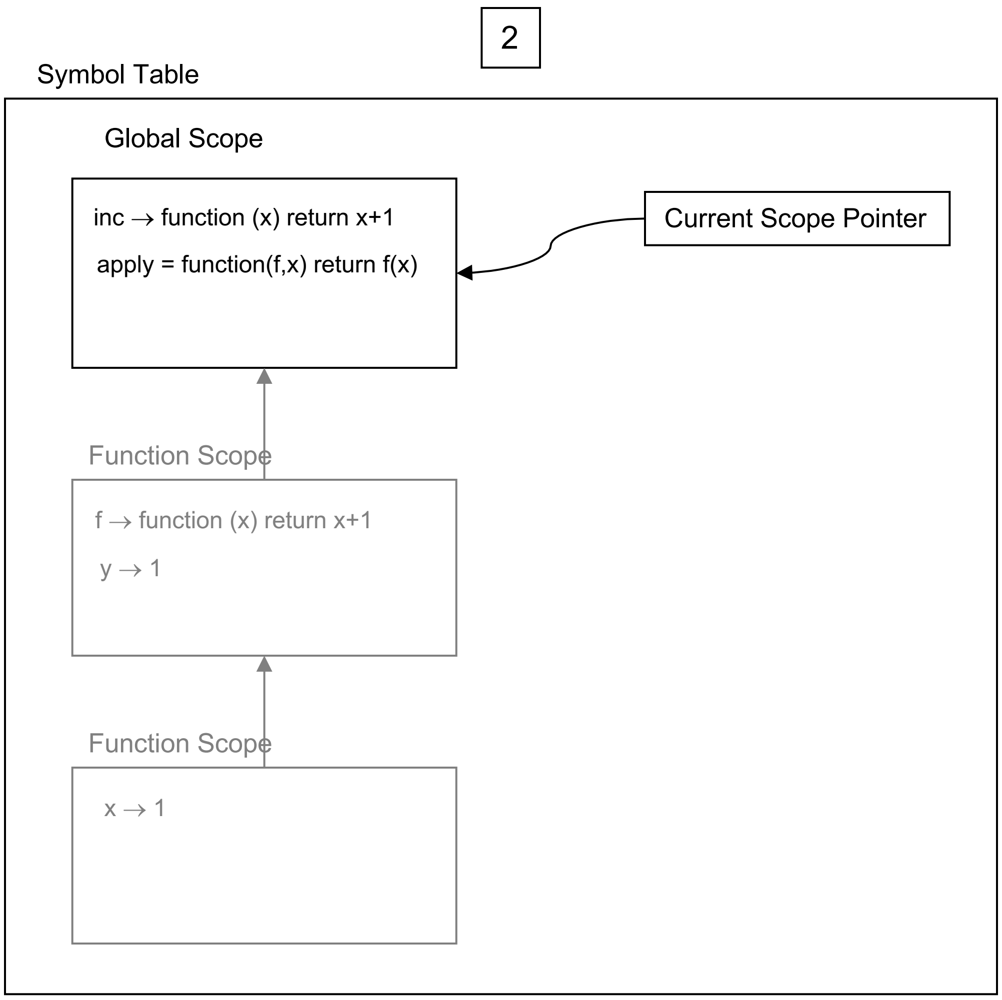
Interpreting Functions




```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

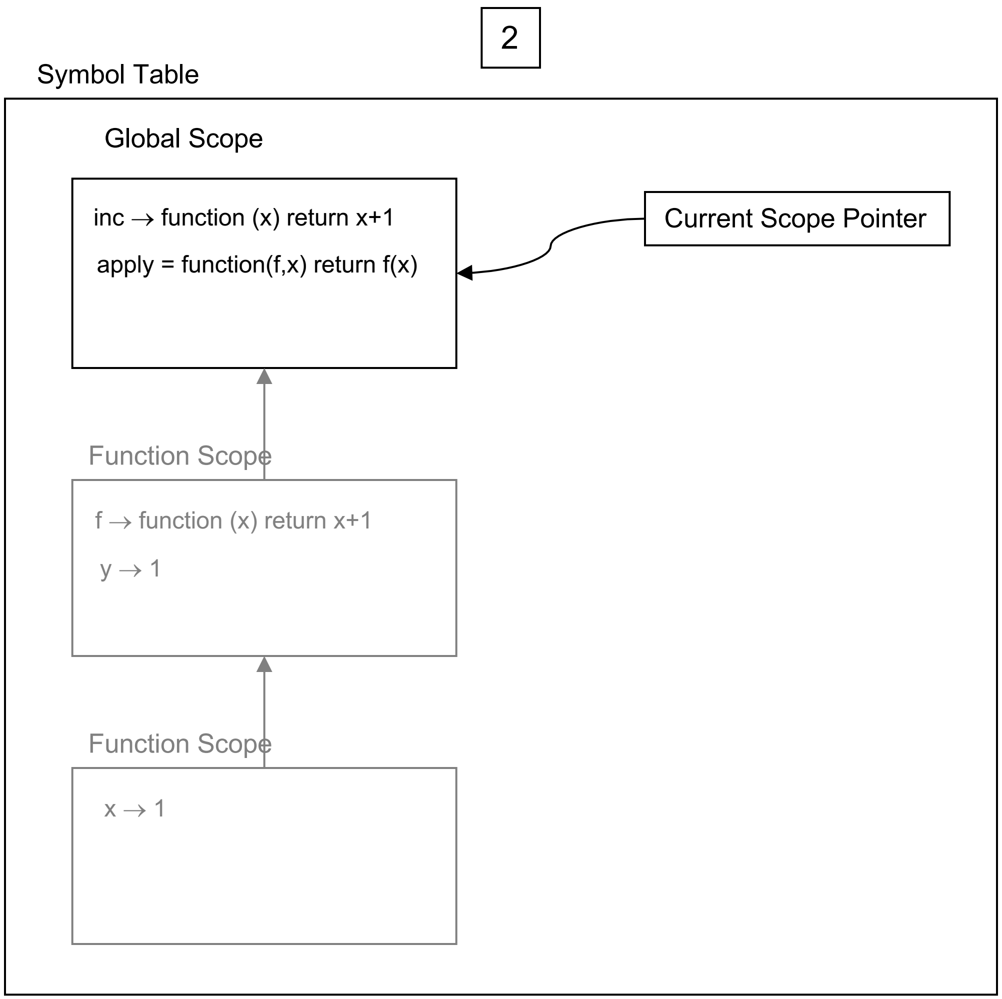
function(f,y) return f(y);

Interpreting Functions




```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

Interpreting Functions



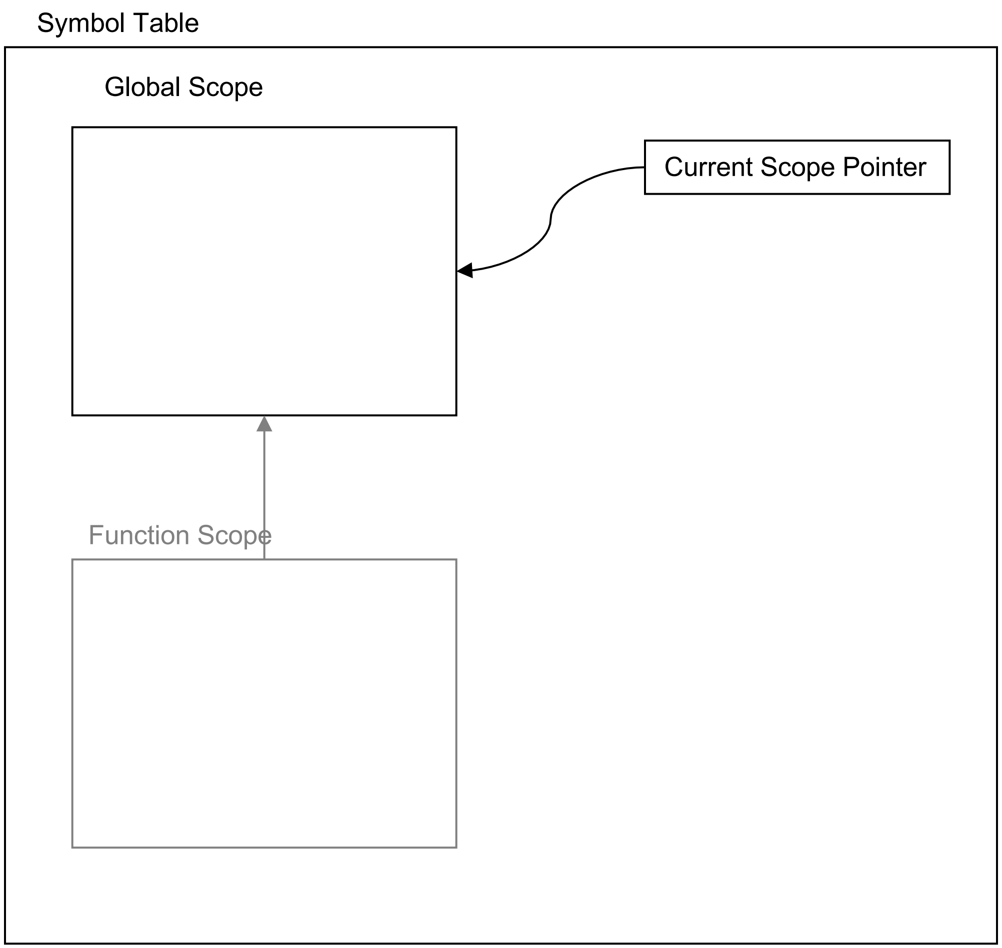

```
declare inc = function (x) return x+1;
declare apply = function(f,y) return f(y);

put apply(inc,1);
```

Higher Order Programming

- This is the hallmark of *higher order programming*
- Definition: *In higher order programming we can pass functions to other functions or return functions as return values from functions.*

```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
  if (q == 0)
      return f;
  else
      return g;
}

put select(inc,dec,1) (3);
```

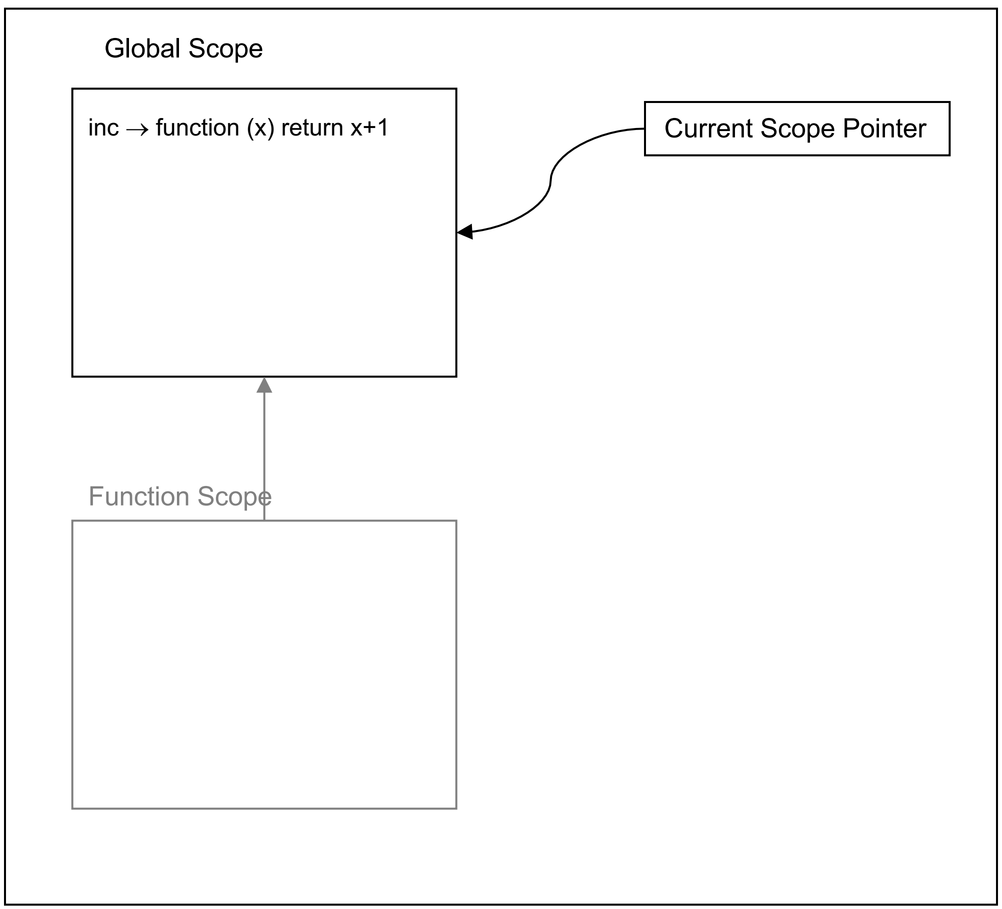
Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

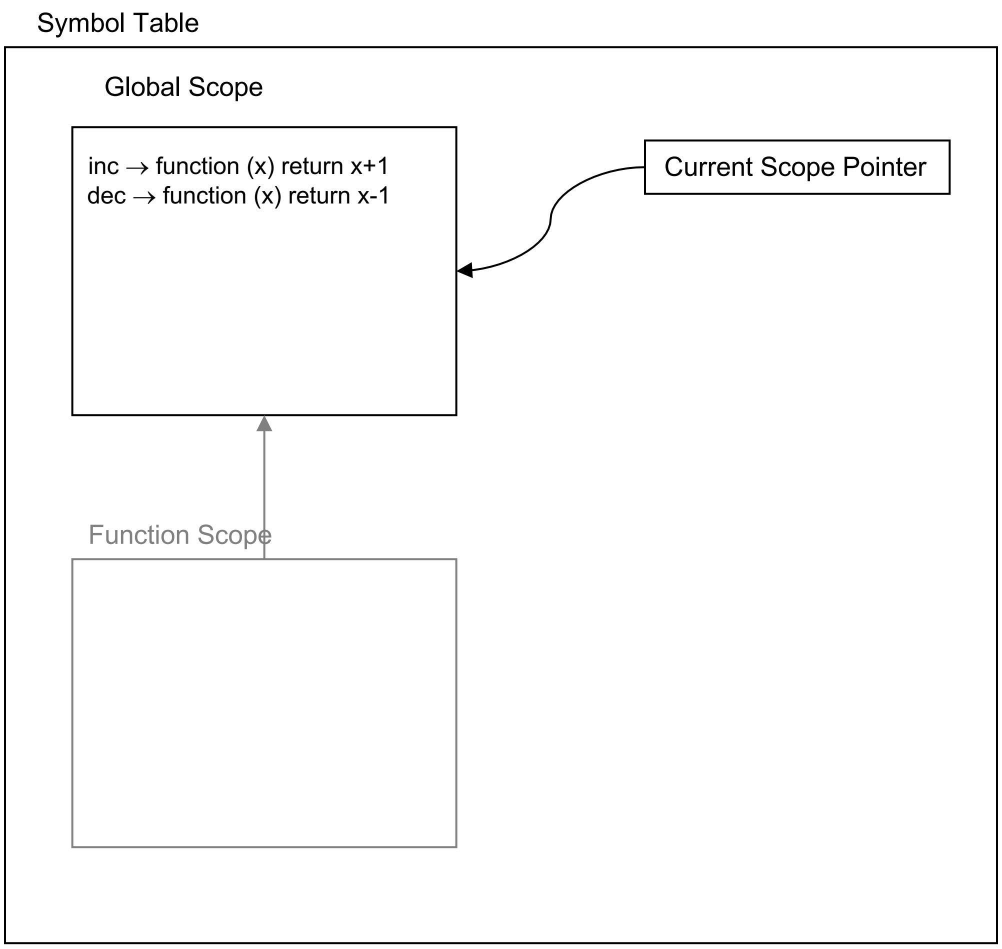
Interpreting Functions

Symbol Table



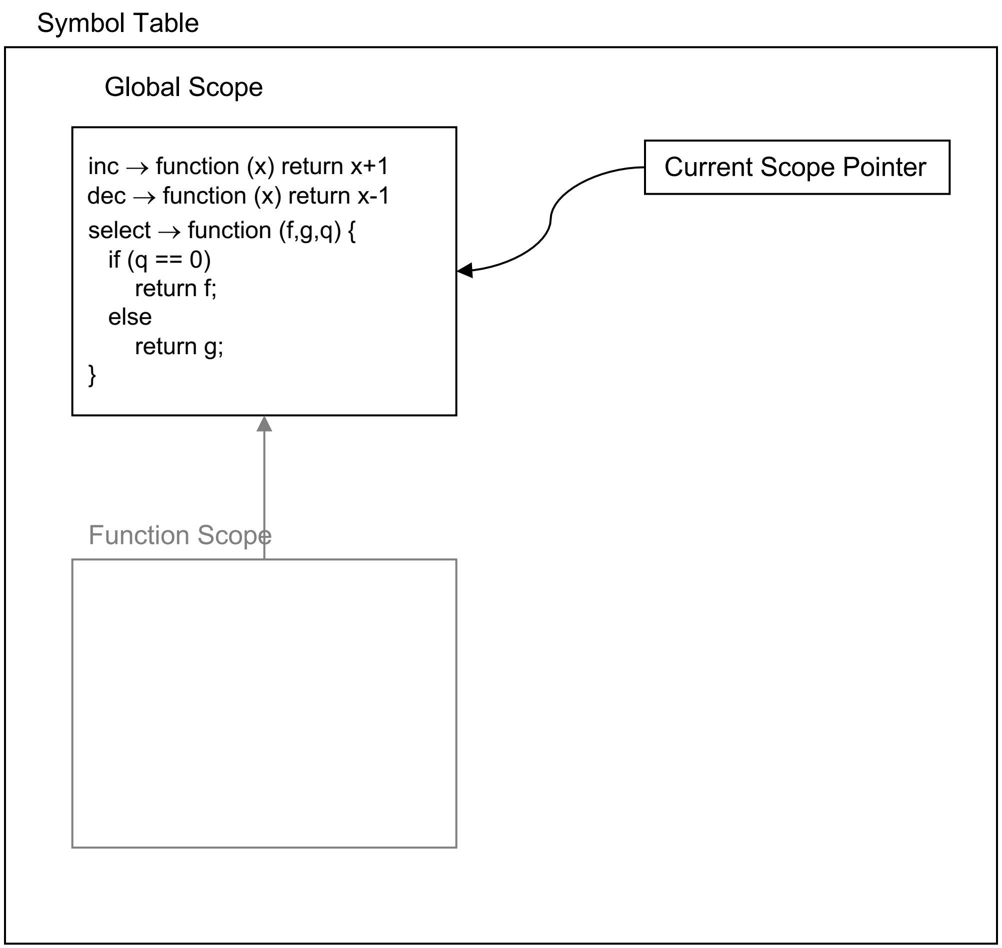

```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

Interpreting Functions



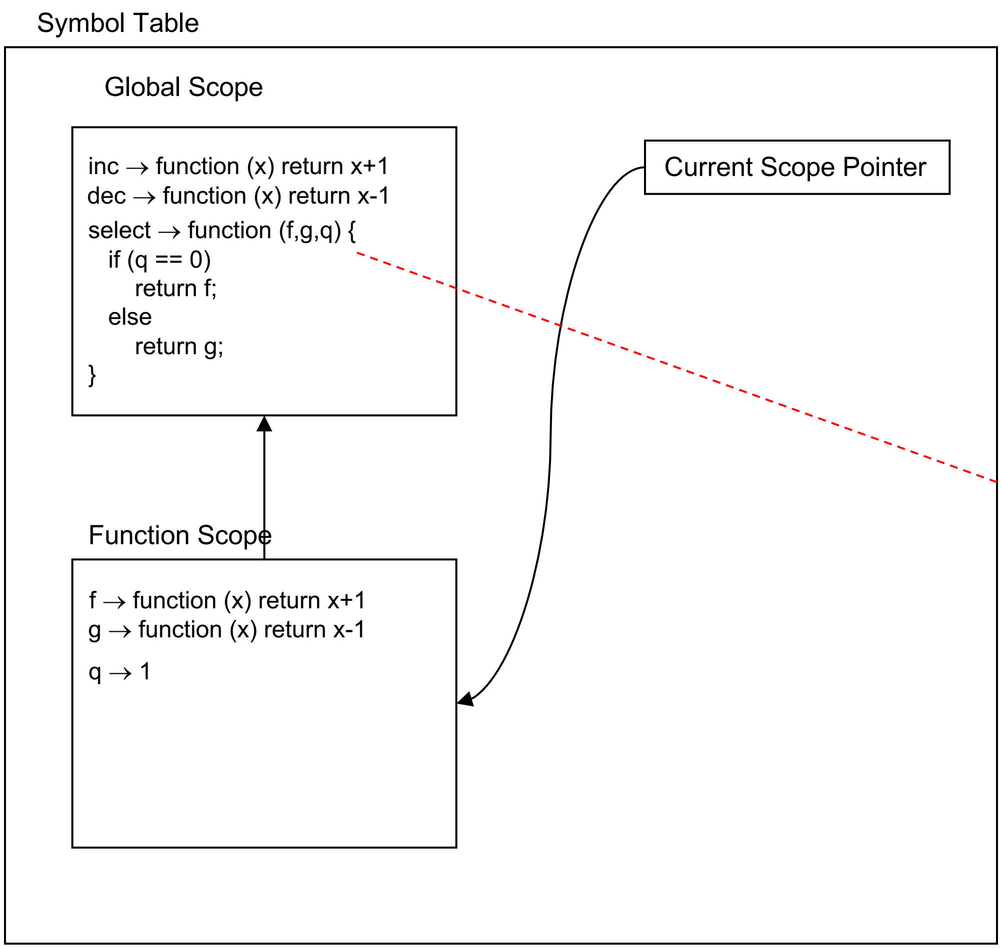

```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
  if (q == 0)
      return f;
  else
      return g;
}

put select(inc,dec,1) (3);
```

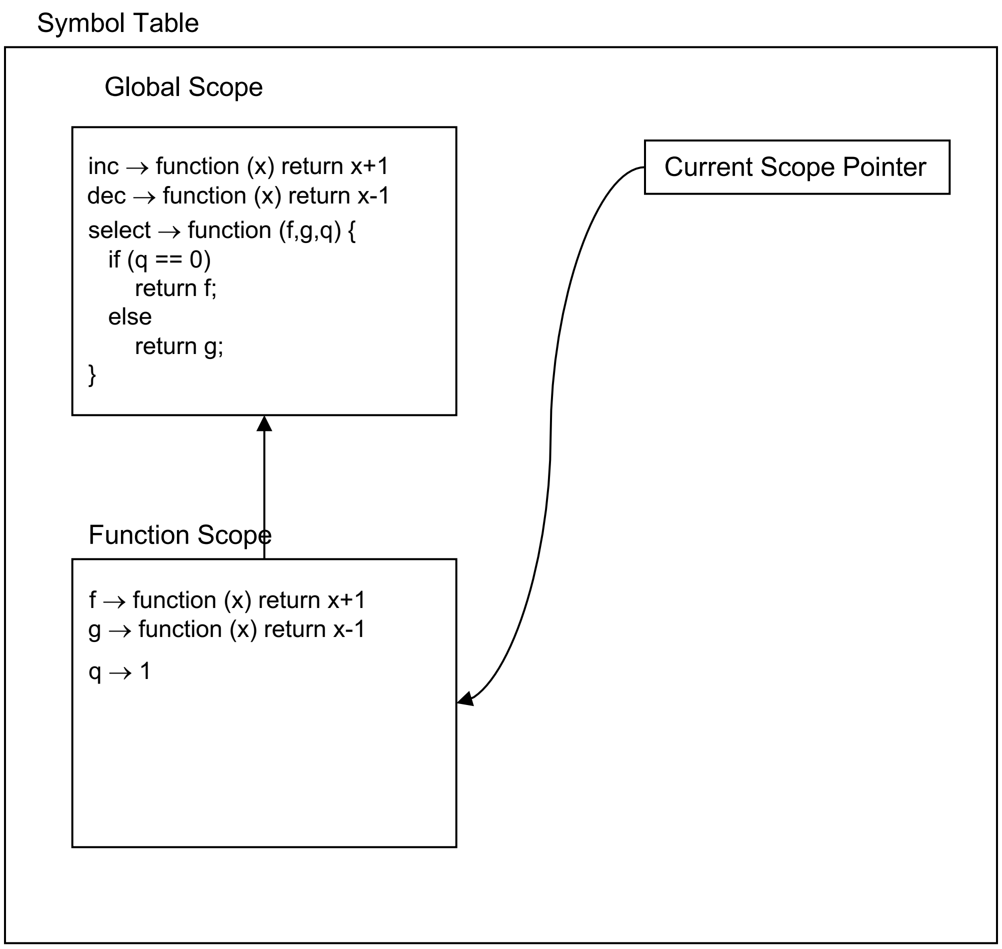
Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
      return f;
   else
      return g;
}

put select(inc,dec,1) (3);
```

```
function (f,g,q) {
   if (q == 0)
      return f;
   else
      return g;
}
```

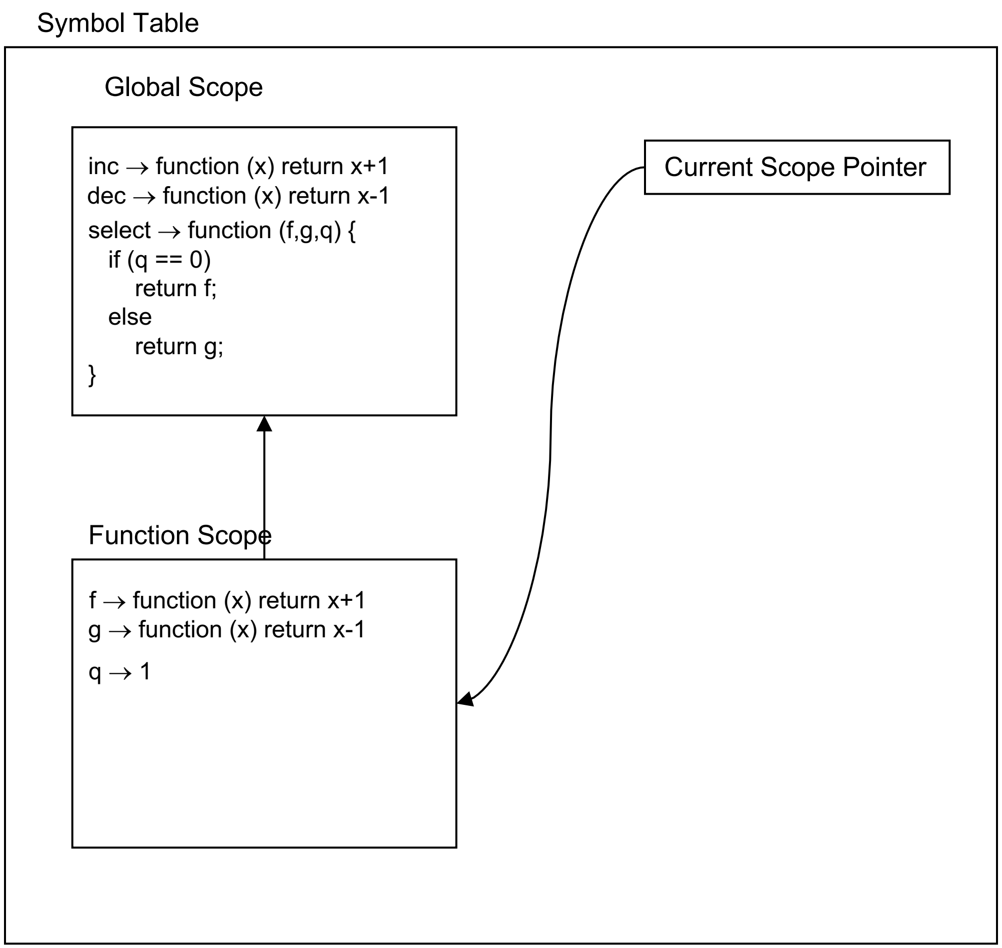
Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

```
function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}
```

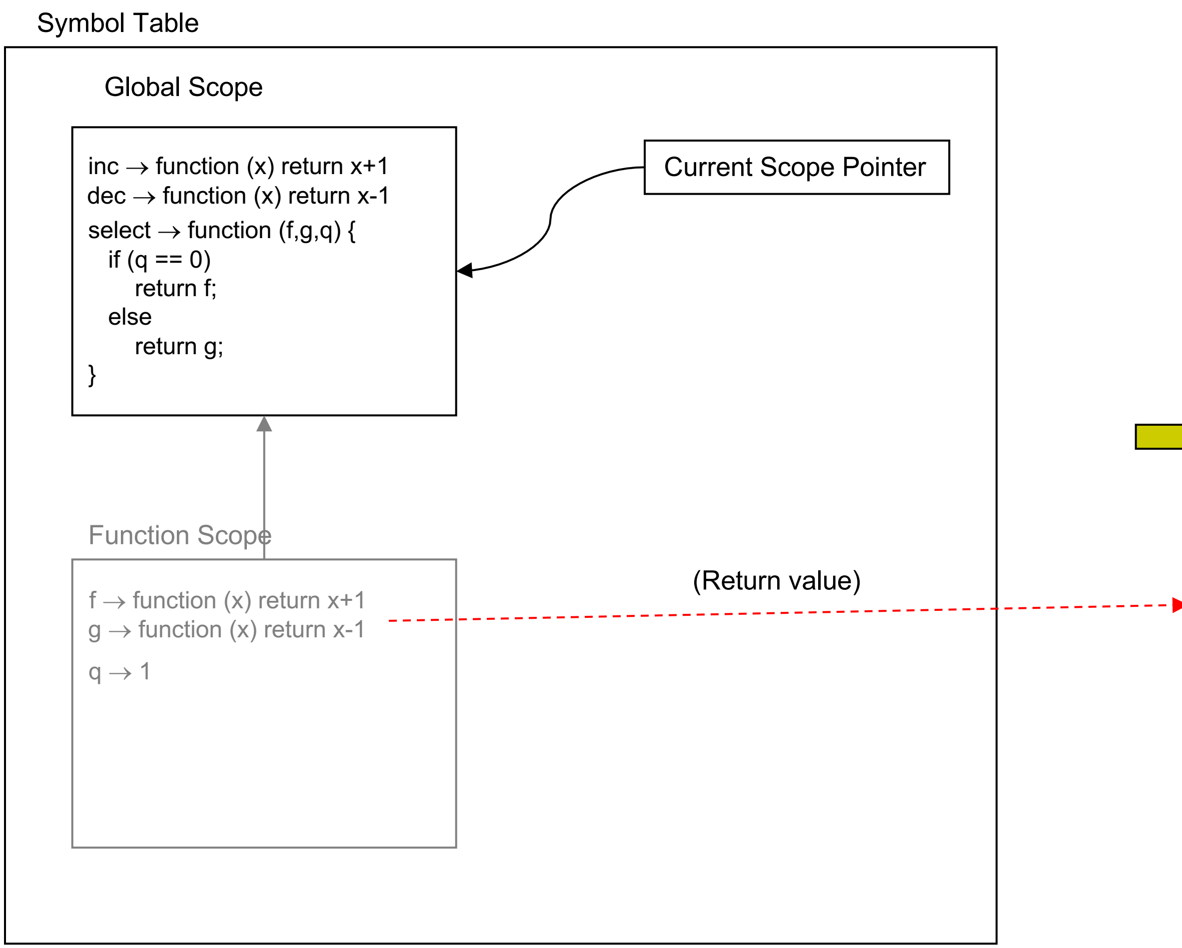
Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

```
function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}
```

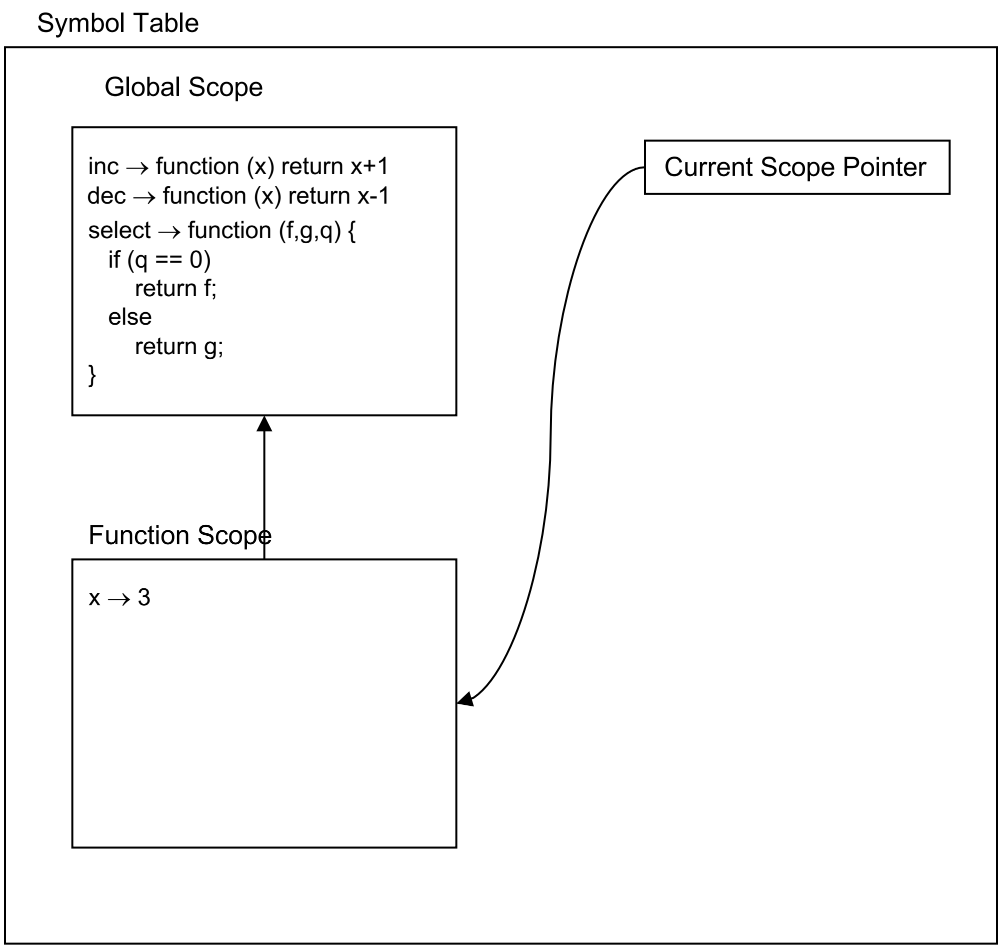
Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

function (x) return x-1;

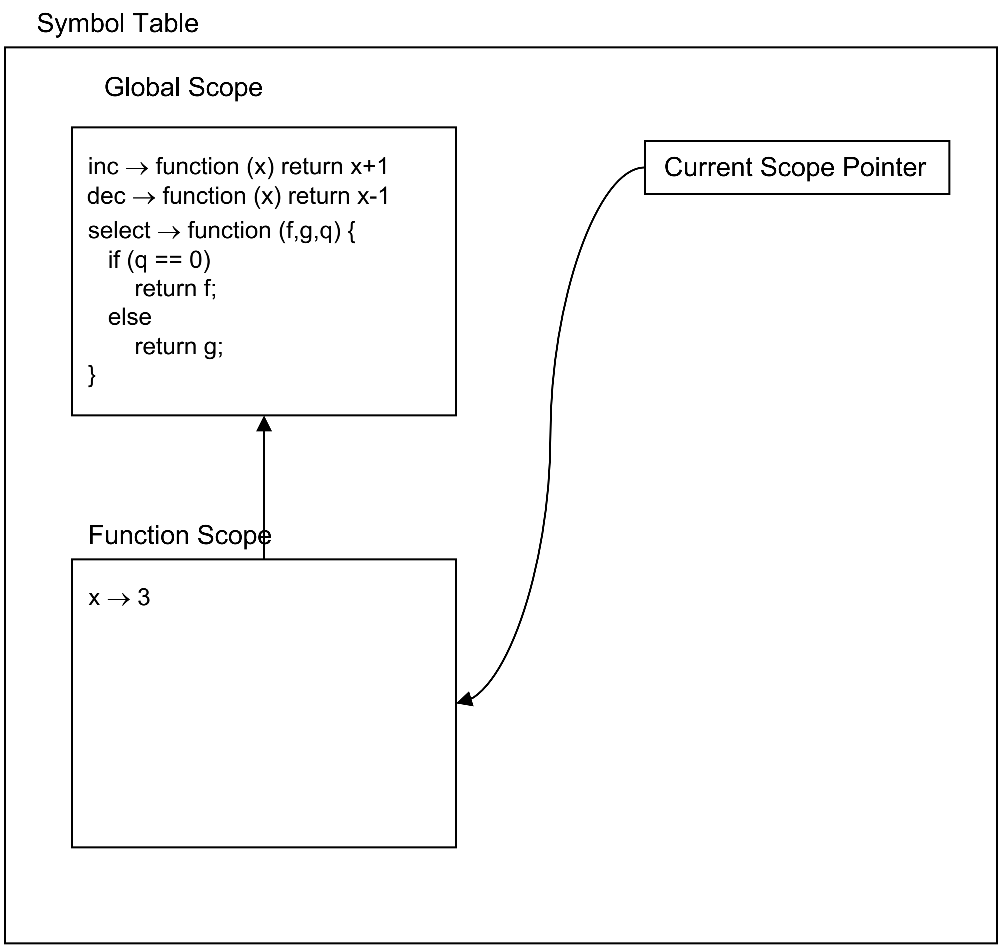
Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

```
function (x) return x-1;
```

Interpreting Functions

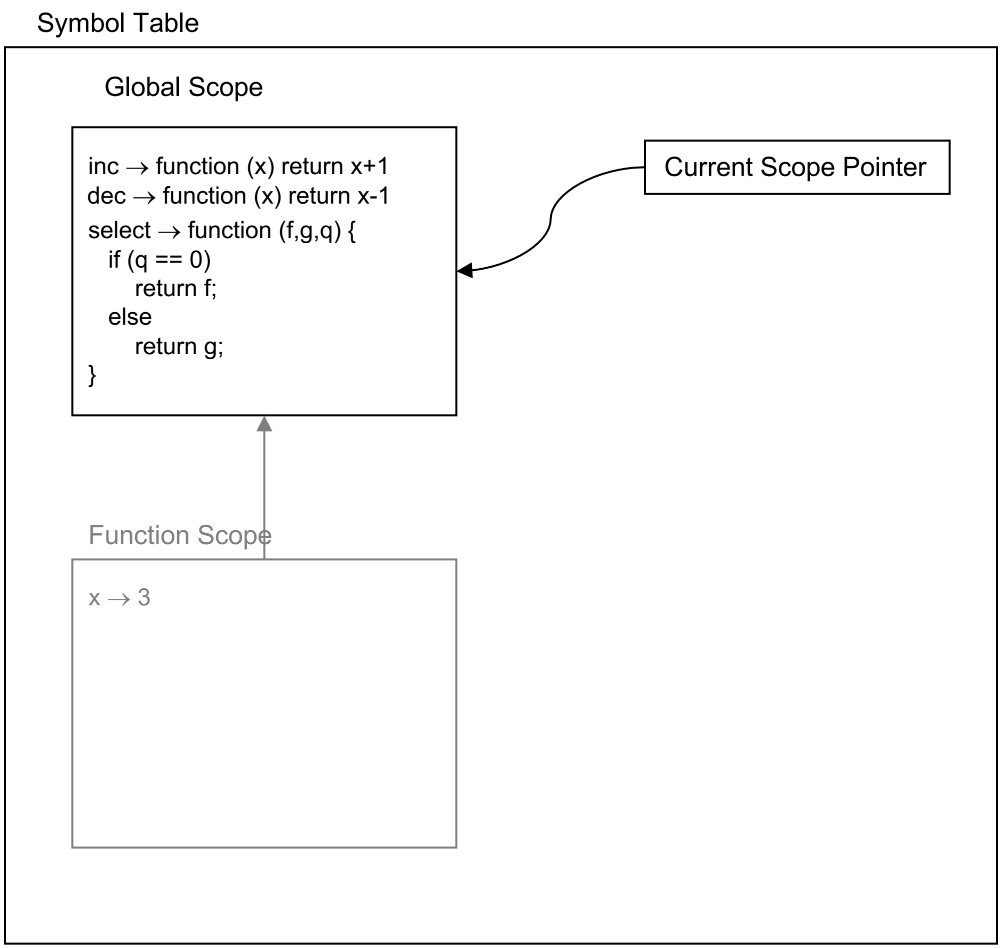



```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(inc,dec,1) (3);
```

```
function (x) return x-1;
```

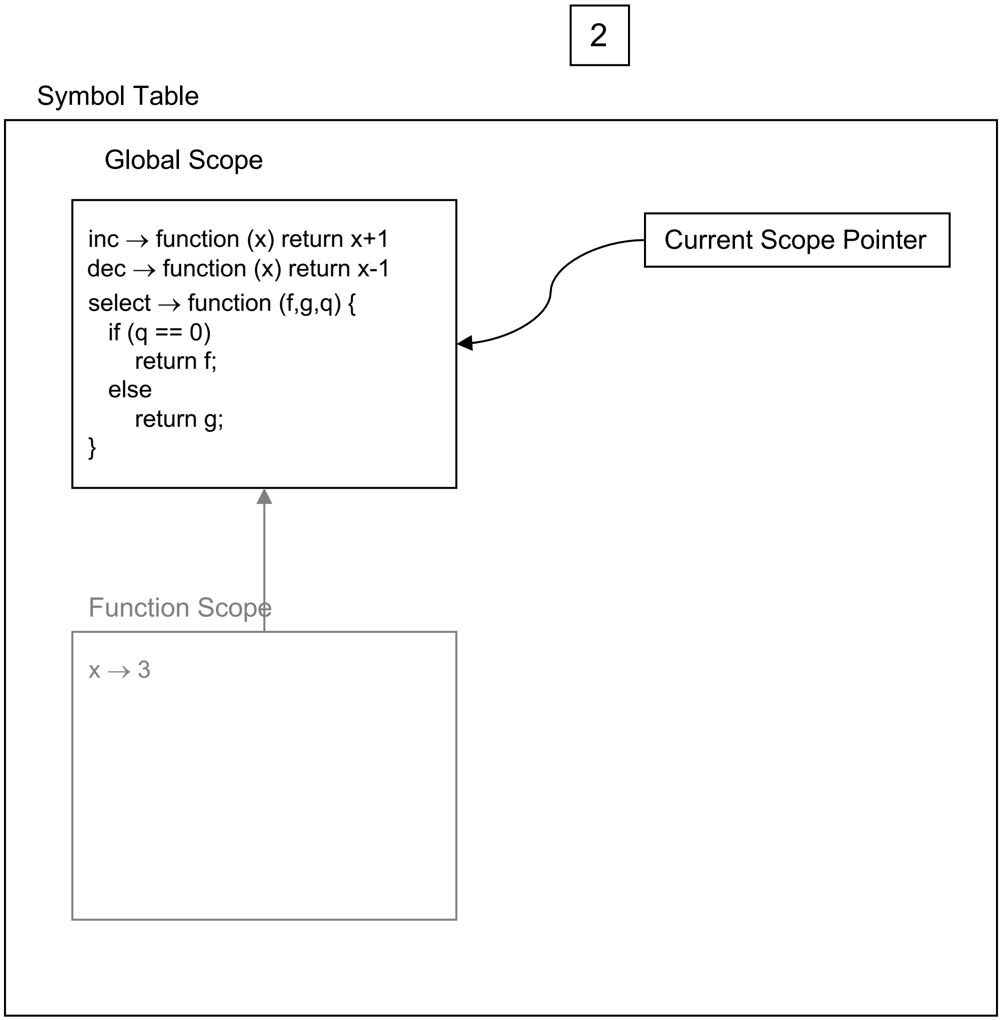
Interpreting Functions

2




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
  if (q == 0)
      return f;
  else
      return g;
}

put select(inc,dec,1) (3);
```

Interpreting Functions




```
declare inc = function (x) return x+1;
declare dec = function (x) return x-1;
declare select = function (f,g,q) {
  if (q == 0)
     return f;
  else
     return g;
}

put select(inc,dec,1) (3);
```

Higher Order Programming

- Why is this interesting?
- It is interesting because higher order programming lets us construct *generic functions* whose behavior can be specialized by passing them more specific functions.
- Consider the function ‘select’ from the previous slide.
- We could use ‘select’ in many different contexts

```
declare mul = function (x) return x*2;
declare div = function (x) return x/2;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(mul,div,1) (4);
```

Higher Order Programming

- **Observation**: In higher order programming we can now have a cascade of function applications
- Consider the last line of the program below:

put select (mul,div,1) (4)

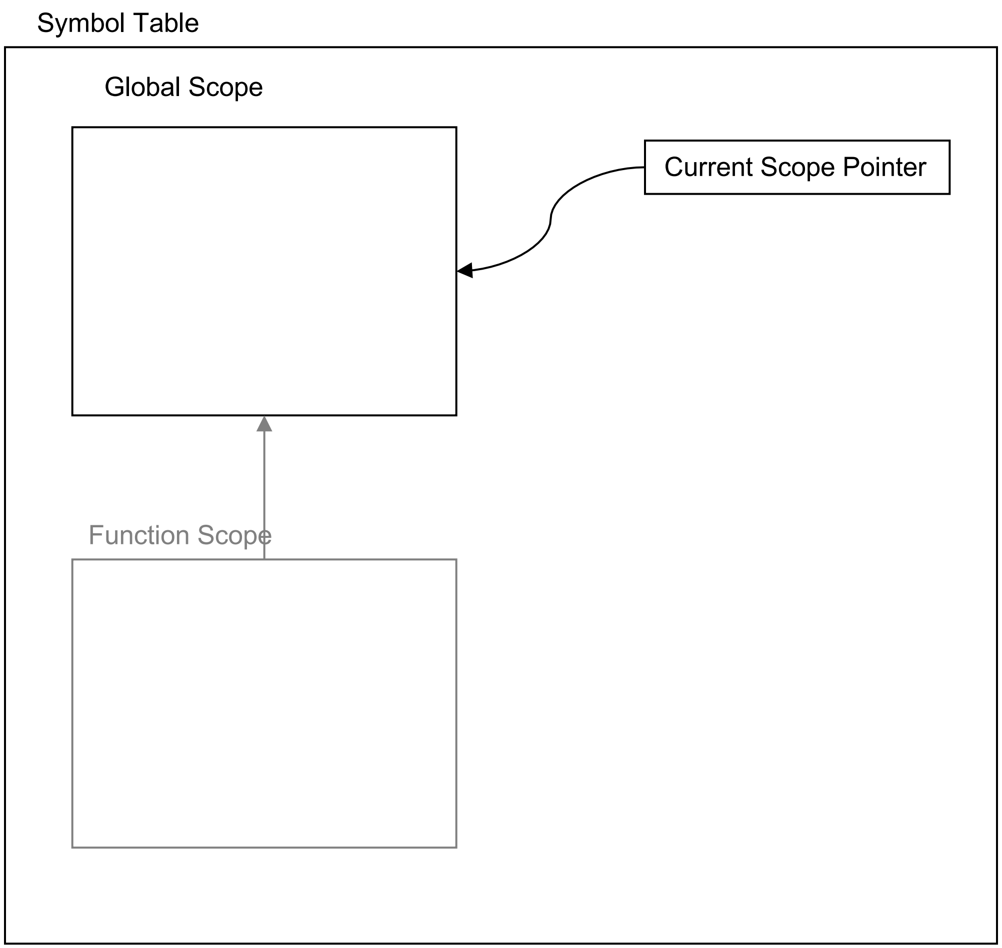
```
declare mul = function (x) return x*2;
declare div = function (x) return x/2;
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select(mul,div,1) (4);
```

Anonymous Functions

- If variables are just handles to access function behavior why not just program with the function behavior directly?
- Consider:

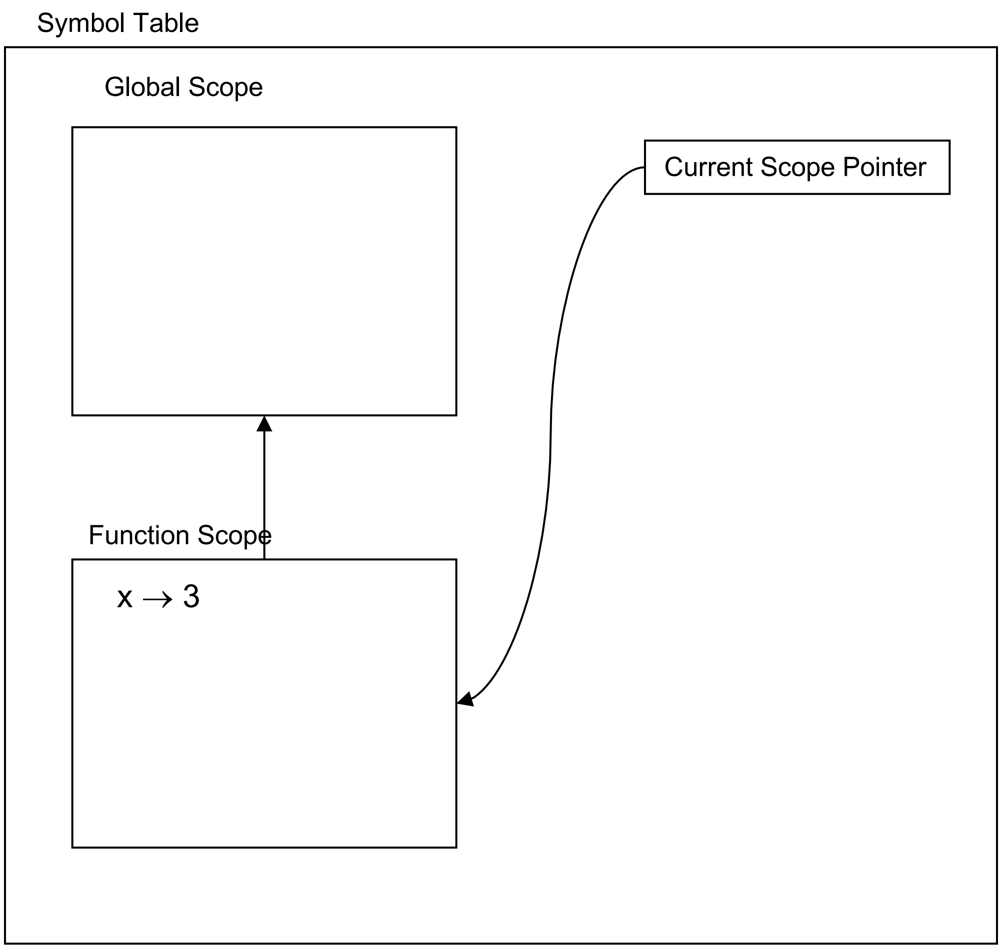
$$\text{put } \underbrace{\text{(function (x) return x+1)}}_{\text{Anonymous Function}} \text{ (3);}$$

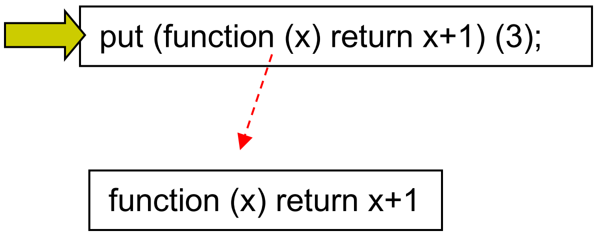
Interpreting Functions



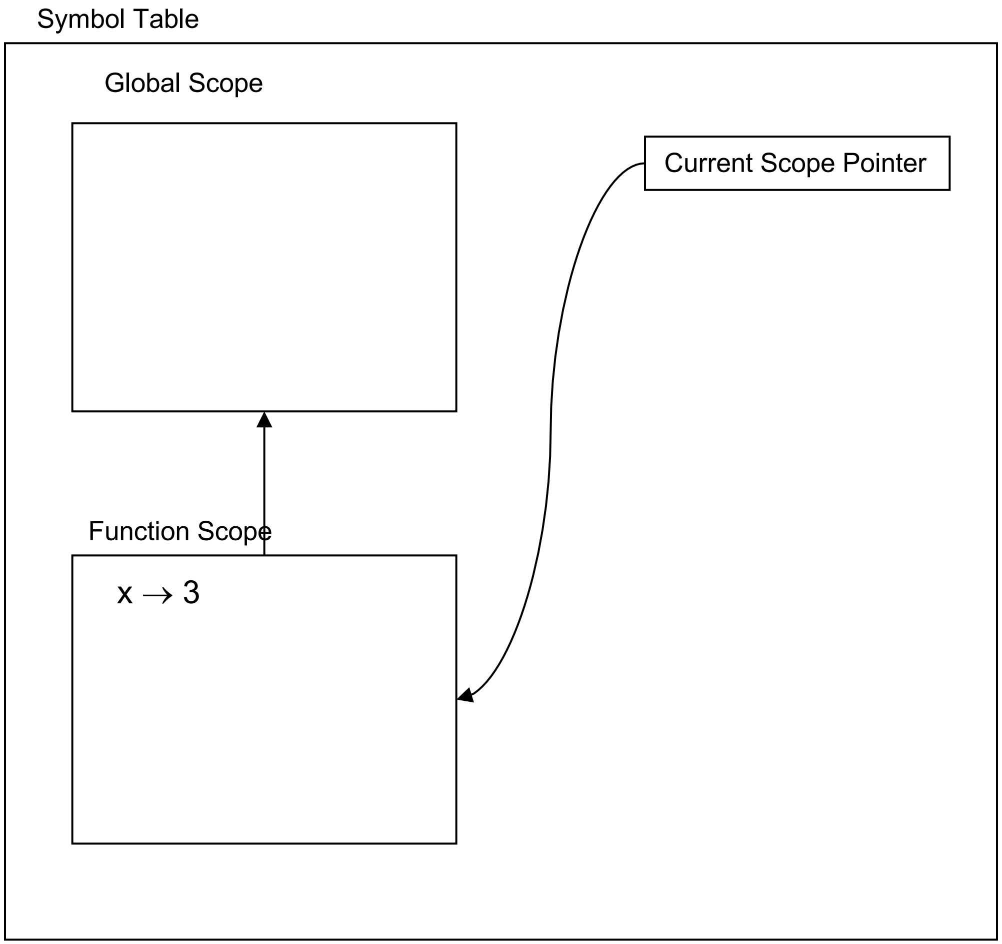
put (function (x) return x+1) (3);

Interpreting Functions





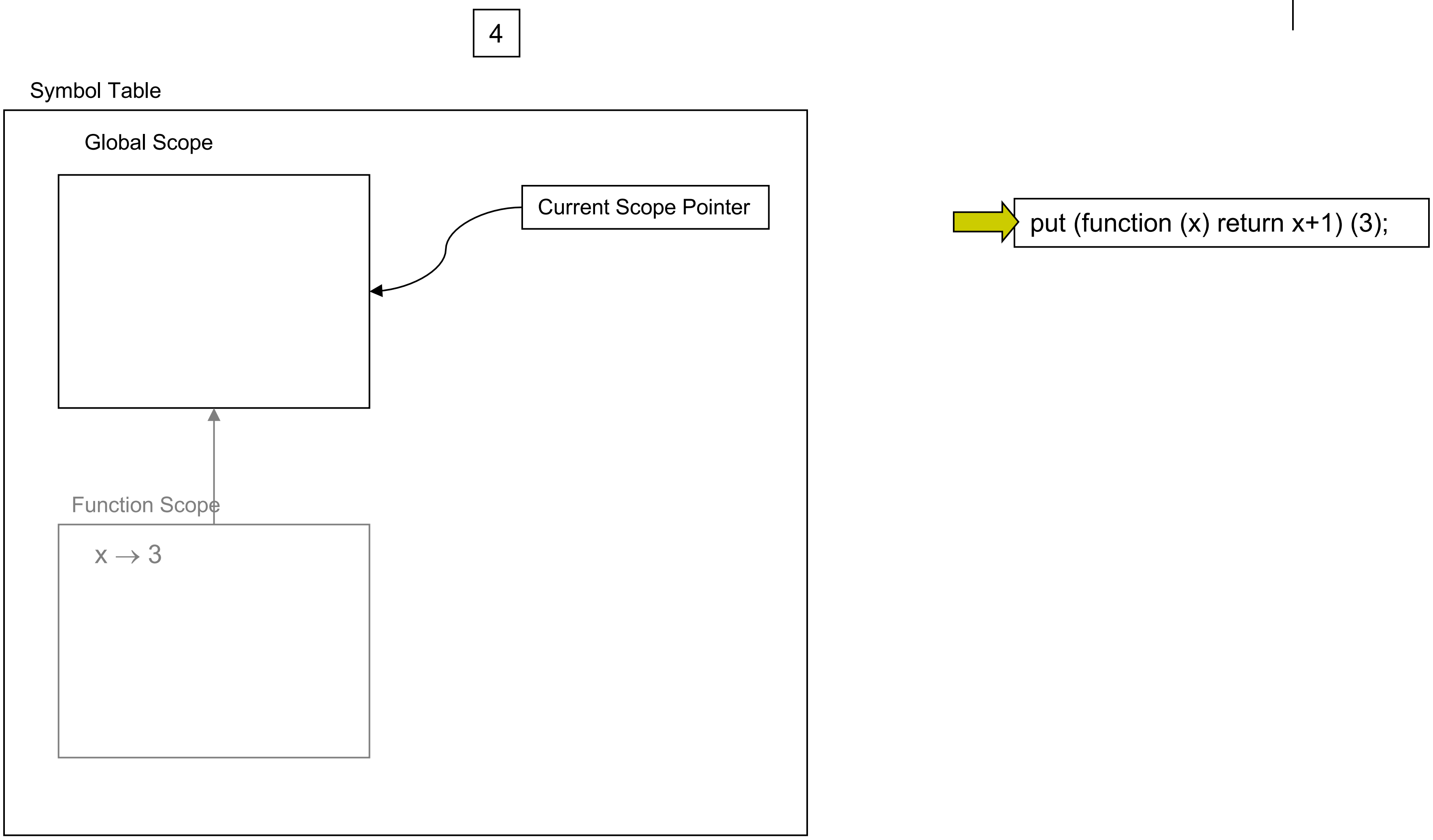
Interpreting Functions



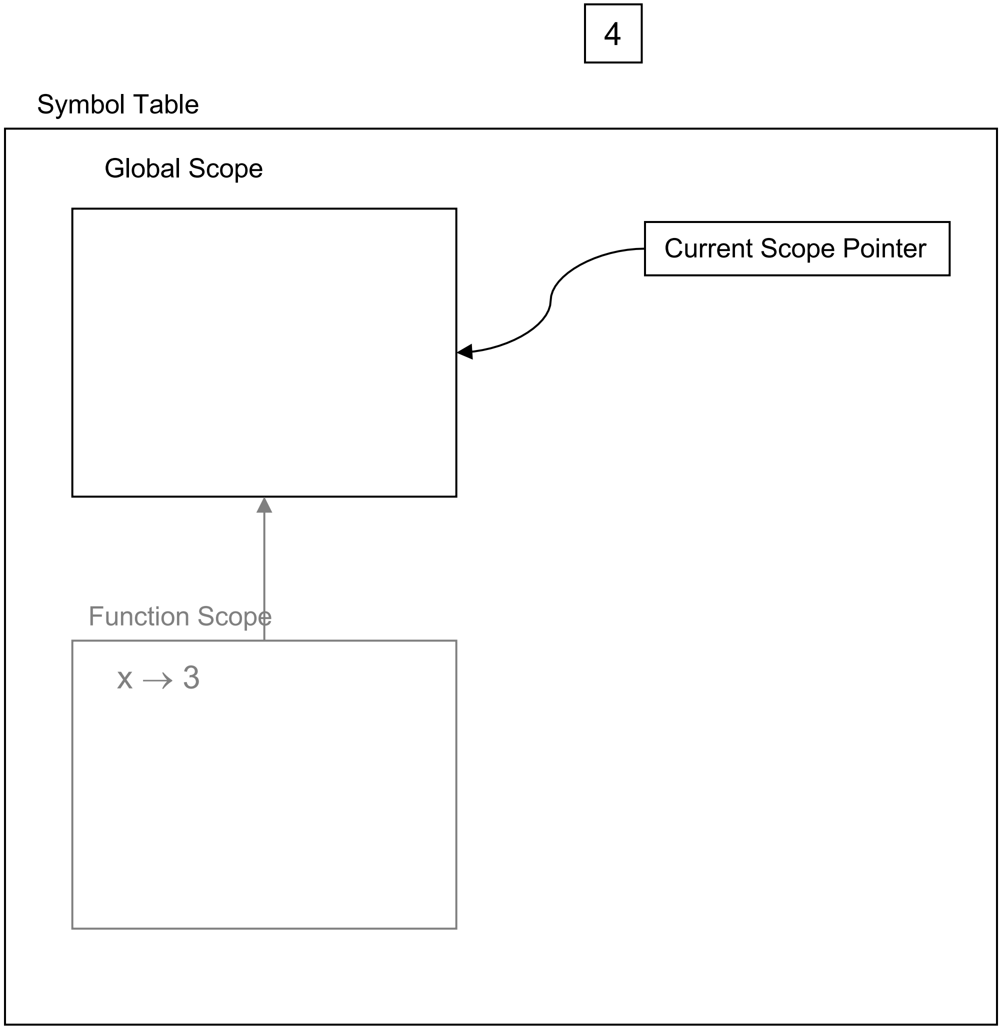
put (function (x) return x+1) (3);

function (x) return x+1

Interpreting Functions



Interpreting Functions



put (function (x) return x+1) (3);

Anonymous Functions

- What does the following program do?

```
declare select = function (f,g,q) {
   if (q == 0)
       return f;
   else
       return g;
}

put select((function (x) return x+1),(function (x) return x-1),1) (3);
```

The Cuppa3h Language

- Similar to Cuppa3 but higher-order.
- Syntactic differences to deal with anonymous functions and function cascades - tuple lists

```
program : stmt_list

stmt_list : stmt stmt_list
          | empty

stmt : DECLARE ID opt_init semi
     | ID '=' exp semi
     | GET ID semi
     | PUT exp semi
     | '(' function_value ')' tuple_list semi
     | ID tuple_list semi
     | RETURN opt_exp semi
     | WHILE '(' exp ')' stmt
     | IF '(' exp ')' stmt opt_else
     | '{' stmt_list '}'

tuple_list : '(' opt_tuple ')' tuple_list
           | '(' opt_tuple ')'

opt_tuple : tuple
          | empty

tuple : exp ',' tuple
      | exp

opt_formal_args : formal_args
                | empty

formal_args : ID ',' formal_args
            | ID

opt_init : '=' exp
         | empty

opt_exp : exp
        | empty

opt_else : ELSE stmt
         | empty

semi : ';'
     | empty
```

The Cuppa3h Language

```
exp : exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | exp EQ exp
    | exp LE exp
    | INTEGER
    | function_value
    | exp tuple_list
    | ID
    | '(' exp ')'
    | MINUS exp %prec UMINUS
    | NOT exp

function_value : FUNCTION '(' opt_formal_args ')' stmt
```

- Note: function values are just values appearing in expressions.

Cuppa3h

- The addition of *function values* now raises the possibility to write syntactically correct programs that semantically do not make any sense.
- Consider:

```
declare z = function (x) return x+1;

put z+1; // ???
```

☞ At runtime we need to reject the expression ‘z+1’ since adding 1 to a function value does not make sense.

☞ Problem: we need to wait until runtime to discover that that is an illegal expression (this is called *dynamic type checking*).

☞Alternative: introduce a *type system* and *static type checking* to discover these kind of illegal expressions before interpretation/compilation begins.

Python is Higher-Order

Untitled — Edited

lutz — python — 80×24

```
Last login: Tue Dec  5 18:02:56 on ttys001
MacBook:~ lutz$ python
Python 3.6.1 |Anaconda 4.4.0 (x86_64)| (default, May 11 2017, 13:04:09)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> (lambda x: x) (lambda y: y+1) (3)
4
>>> 
```