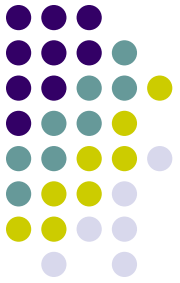




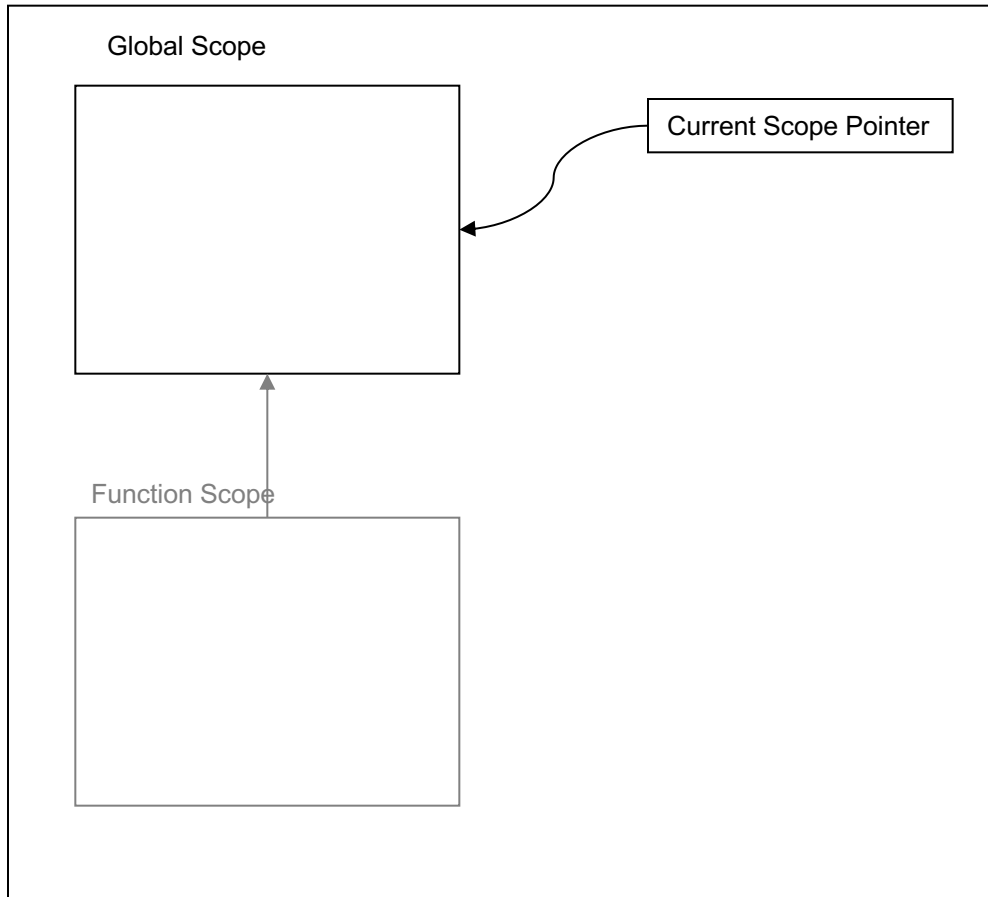
Interpreter Implementation

- The crucial insight to implementing functions is that function names act just like variable names - they are the key into a symbol lookup table.
 - During function declaration we enter the function name into the symbol table
 - During a function call we search for the function name in the symbol table
- The second important insight is that the function body is the value that we store with the function name in the symbol table.
 - During a function call we lookup the function name in the symbol table and return the function body for interpretation.
- The symbol table is extended to distinguish between scalar values and function values

Interpreting Functions



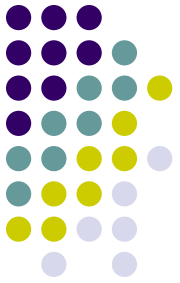
Symbol Table



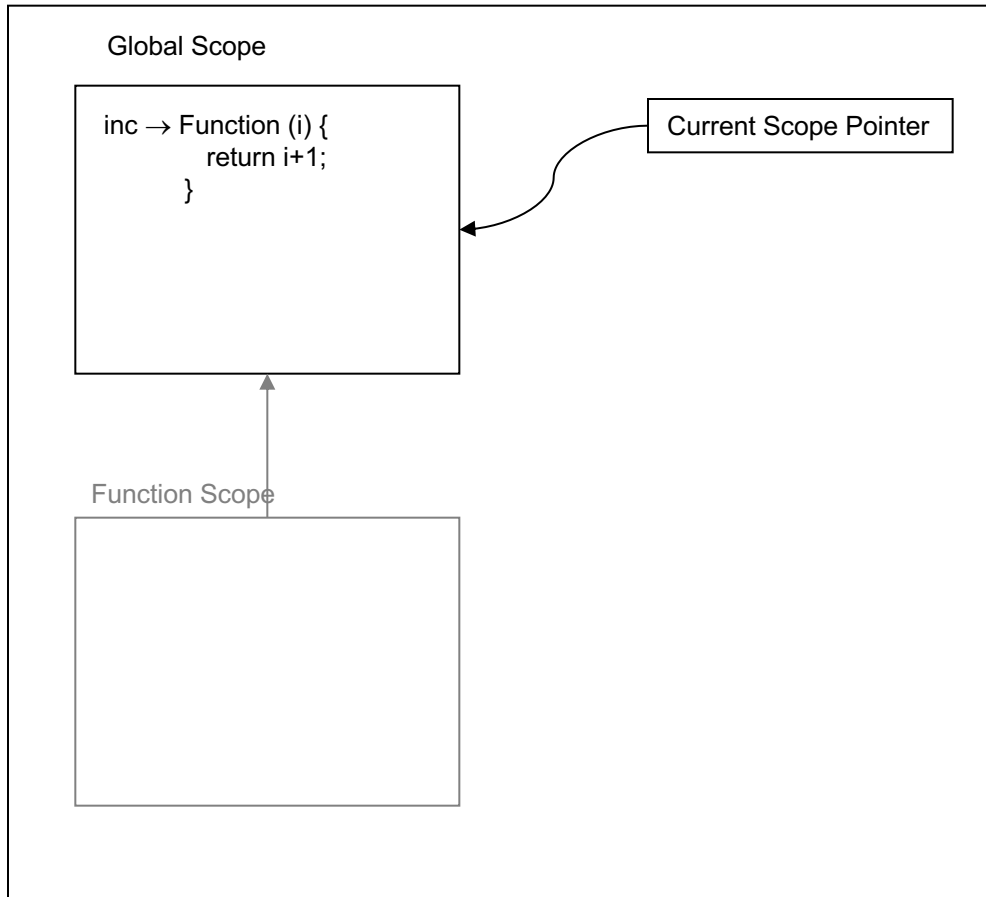
```
declare inc(i) {  
    return i+1;  
}
```

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

Interpreting Functions



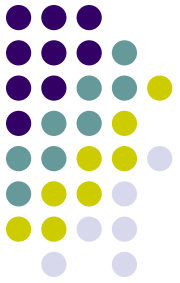
Symbol Table



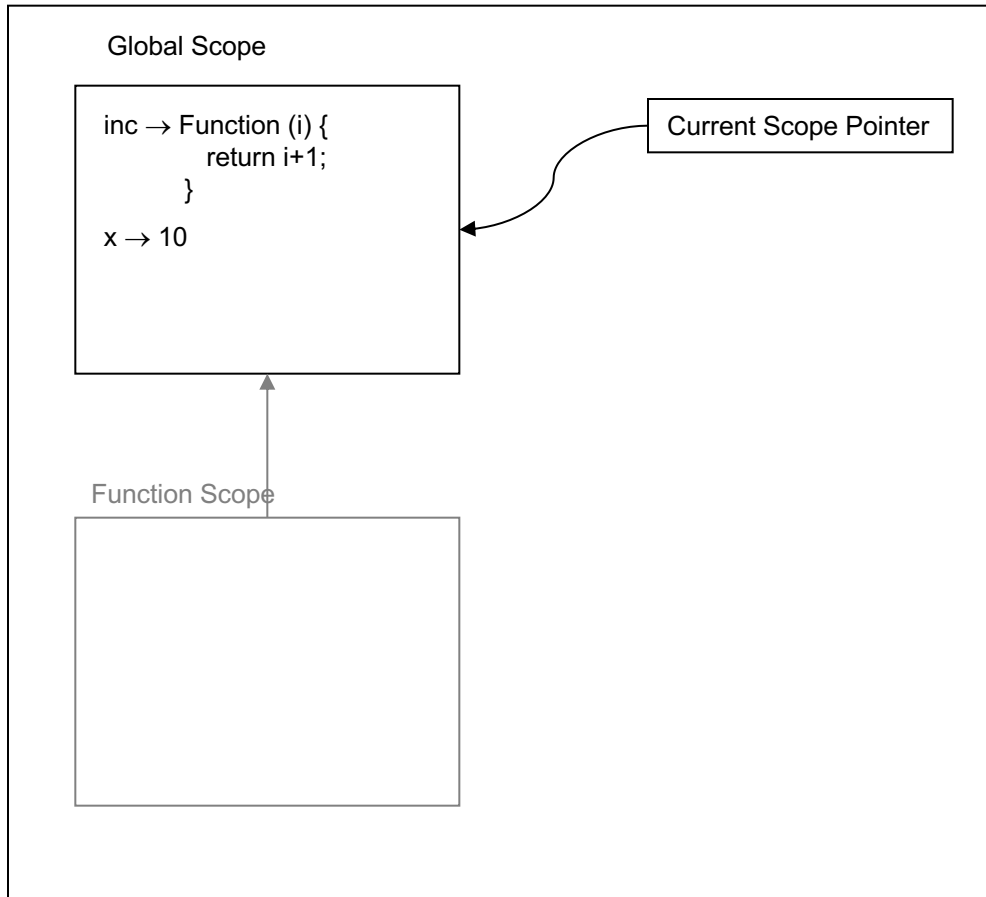
→ declare inc(i) {
 return i+1;
}

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

Interpreting Functions

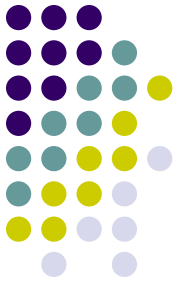


Symbol Table


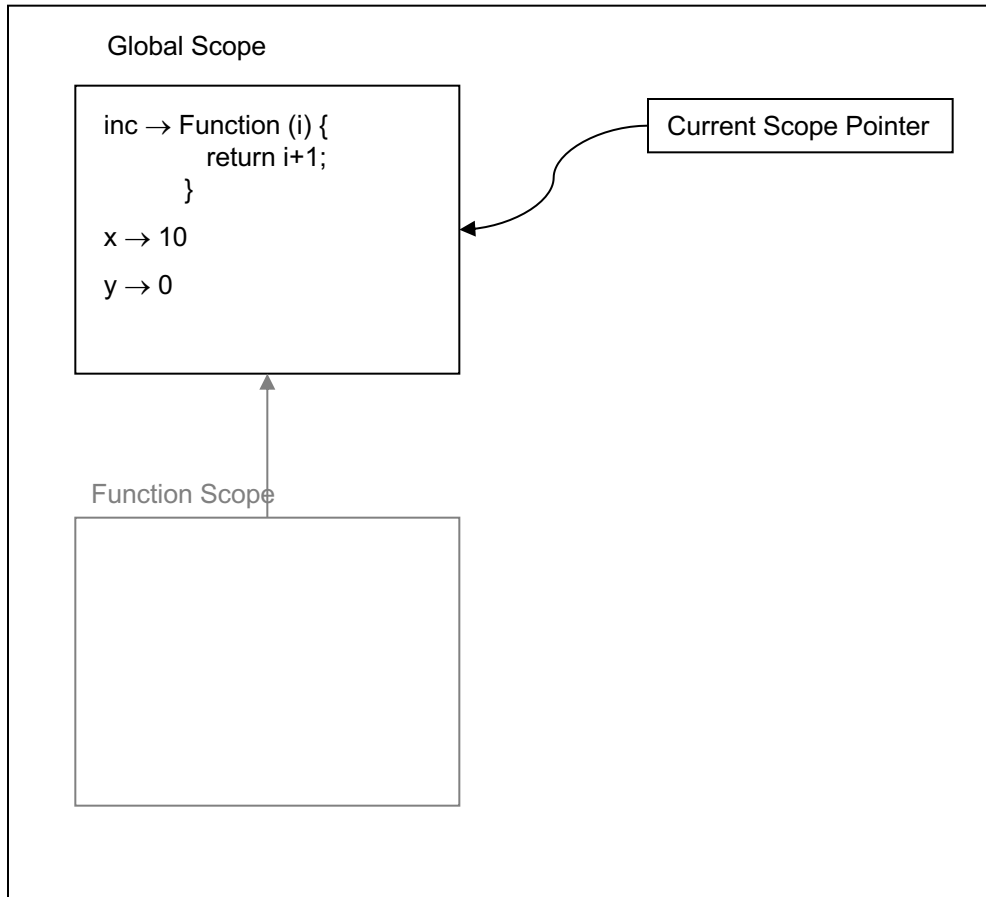


```
declare inc(i) {  
    return i+1;  
}  
  
declare x = 10;  
declare y;  
y = inc(x);  
put y;
```

Interpreting Functions



Symbol Table

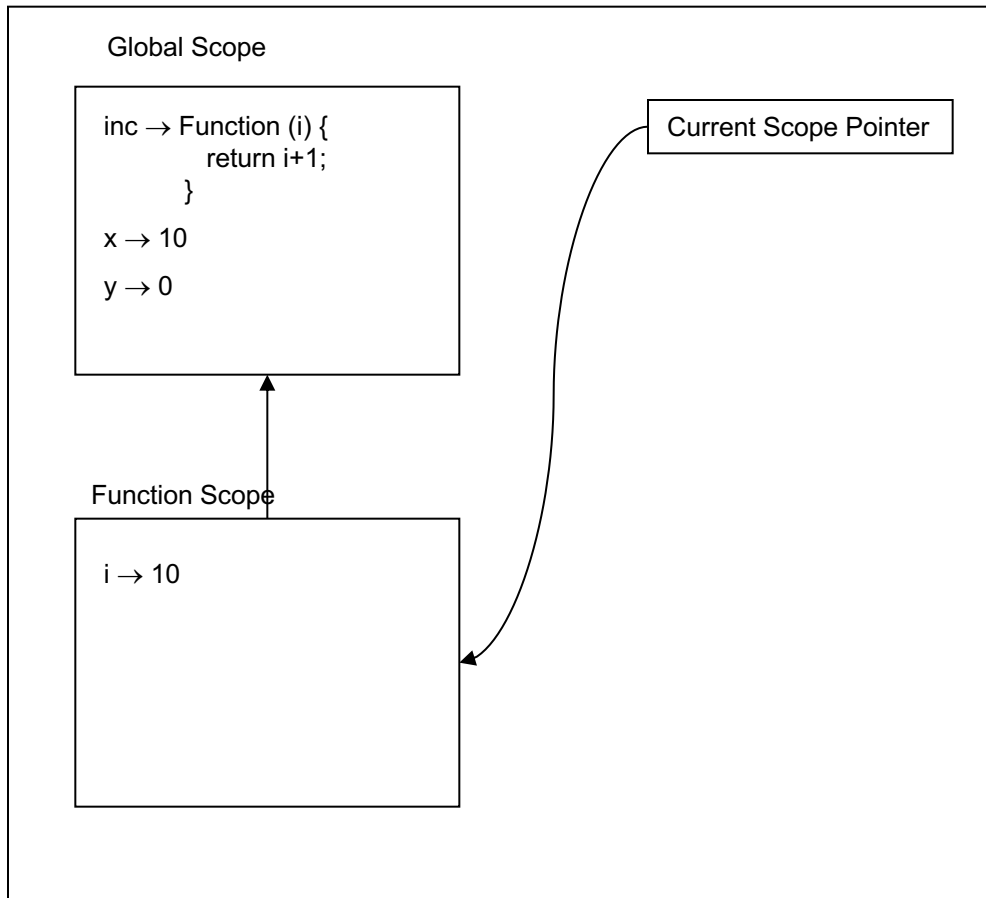


```
declare inc(i) {  
    return i+1;  
}  
  
declare x = 10;  
declare y;  
y = inc(x);  
put y;
```

Interpreting Functions



Symbol Table



declare inc(i) {
 return i+1;
}

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

Function (i) {
 return i+1;
}

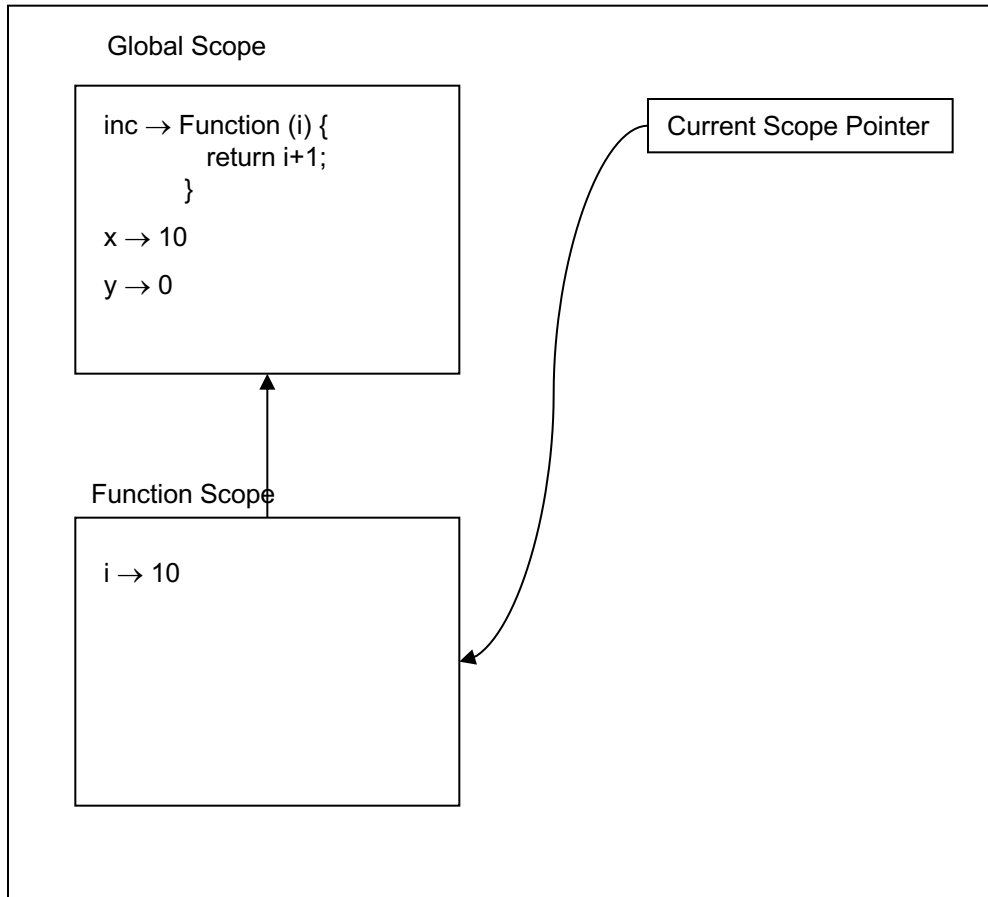
Setup the function call:

- lookup function name
- retrieve function body
- push new function scope
- init formal parameters with actual parameters

Interpreting Functions



Symbol Table



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

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



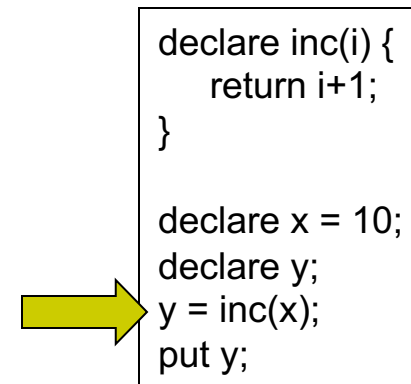
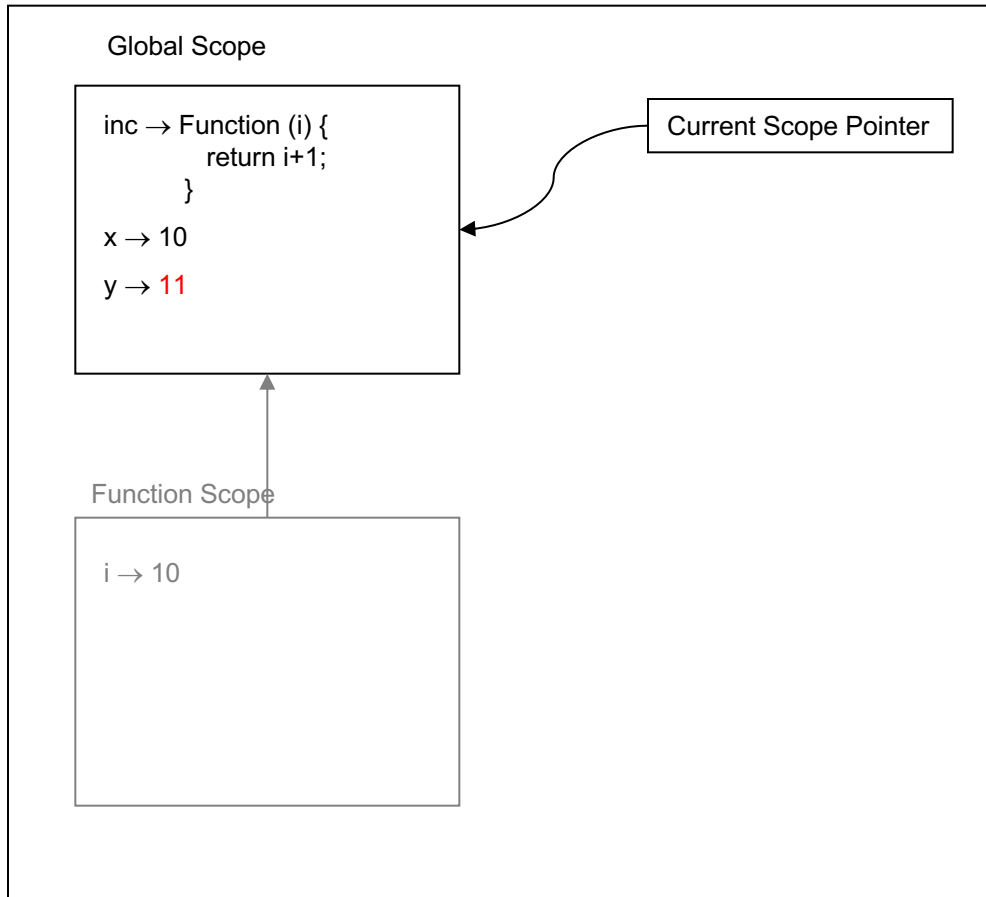
```
Function (i) {  
    return i+1;  
}
```

Execute the called function and compute return value.

Interpreting Functions

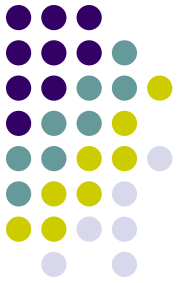


Symbol Table

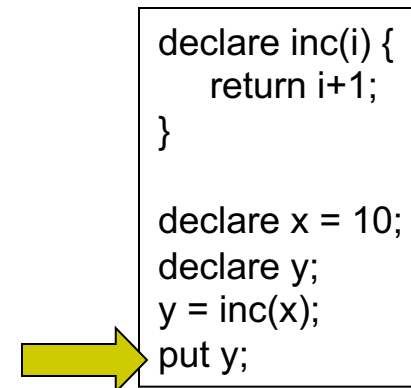
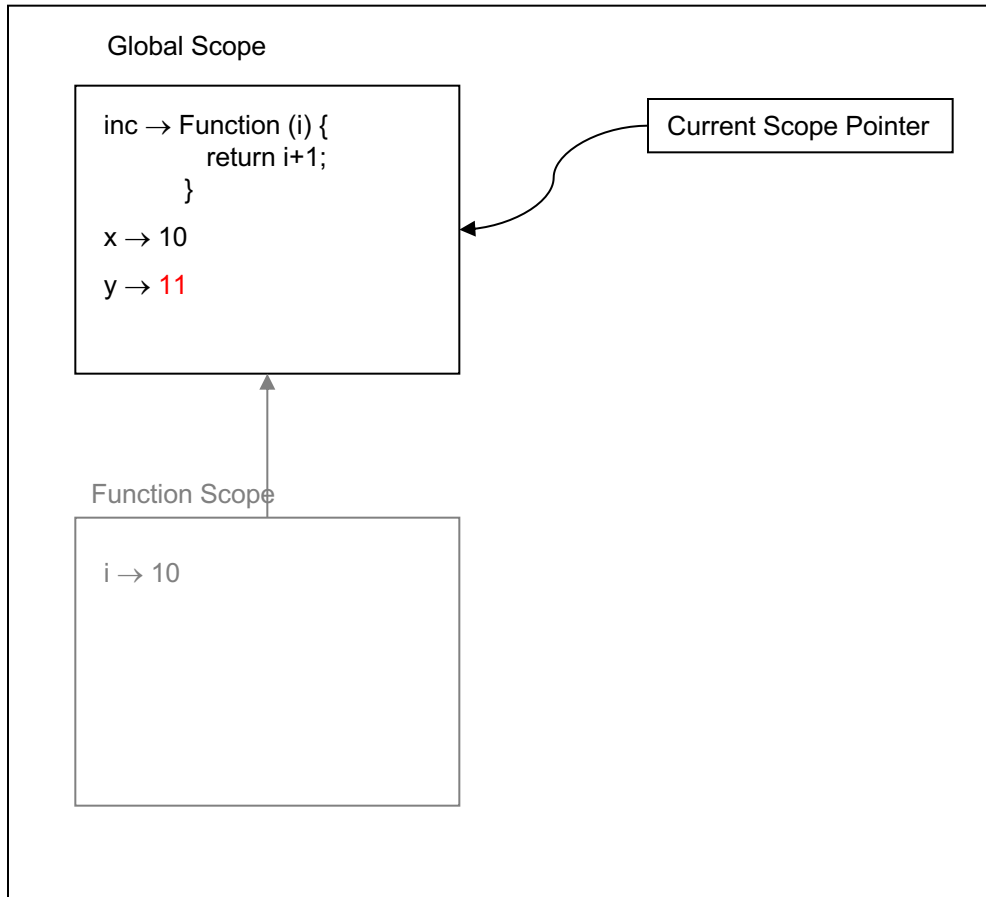


- Exit the called function:
- pop the function scope
 - store the return value in y

Interpreting Functions

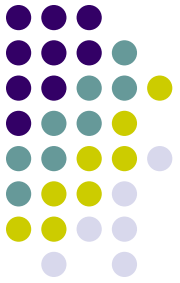


Symbol Table

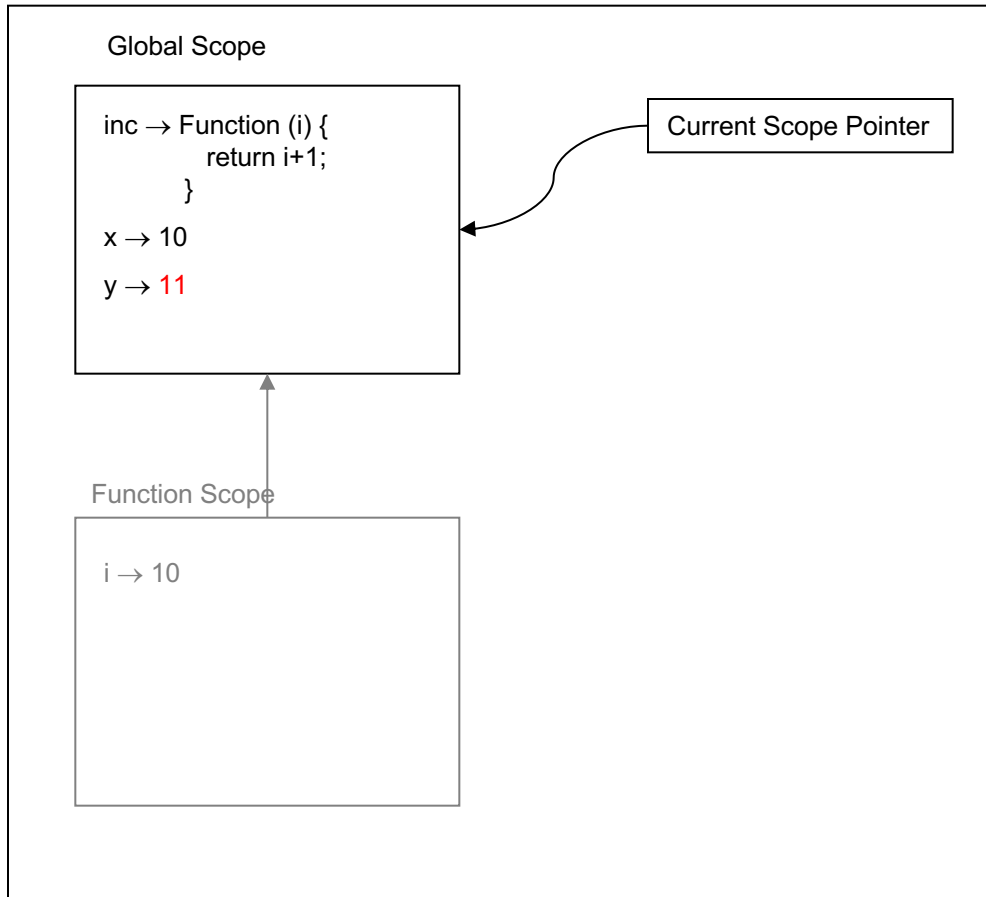


Execute the put statement \Rightarrow 11

Interpreting Functions



Symbol Table



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

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



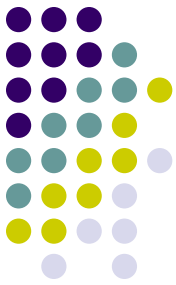
Interpreting Functions

- Note that we use the function value just like we would use the value of a variable, but instead of using it in some arithmetic expression we simply interpret the body of the function in order to compute a return value.

Cuppa3 Frontend

Listing 8.2: An LL(1) grammar for the Cuppa3 language.

```
1  stmt_list : (stmt)*
2
3  stmt : declare ID decl_suffix
4        | ID id_suffix
5        | get ID ;?
6        | put exp ;?
7        | return exp? ;?
8        | while \( exp \) stmt
9        | if \( exp \) stmt (else stmt)?
10       | \{ stmt_list \}
11
12  decl_suffix : \( formal_args? \) stmt
13              | = exp ;?
14              | ;?
15
16  id_suffix : \( actual_args? \) ;?
17            | = exp ;?
18
19  exp : exp_low
20  exp_low : exp_med ((= | =<) exp_med)*
21  exp_med : exp_high ((\+ | -) exp_high)*
22  exp_high : primary ((\* | /) primary)*
23
24  primary : INTEGER
25           | ID (\( actual_args? \))?
26           | \( exp \)
27           | - primary
28           | not primary
29
30  formal_args : ID (, ID)*
31  actual_args : exp (, exp)*
32
33  ID : <any valid variable name>
34  INTEGER : <any valid integer number>
```





Cuppa3 Frontend




- The frontend is like all of our other Cuppa LL(1) frontends
 - we construct an AST using a parser constructed from an LL(1) grammar.
- We will concentrate on the three new features outlined in the previous slide.

Cuppa3 Frontend



```
stmt : declare ID decl_suffix
```

```
decl_suffix : \( formal_args? \) stmt  
            | = exp ;?  
            | ;?
```

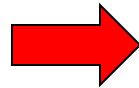
```
def decl_suffix(stream):  
    if stream.pointer().type in ['LPAREN']:  
        stream.match('LPAREN')  
        if stream.pointer().type in ['ID']:  
            args = formal_args(stream)  
        else:  
            args = ('LIST', [])  
        stream.match('RPAREN')  
        body = stmt(stream)  
        return ('FUNCTION', args, body)   
    elif stream.pointer().type in ['ASSIGN']:  
        stream.match('ASSIGN')  
        e = exp(stream)  
        if stream.pointer().type in ['SEMI']:  
            stream.match('SEMI')  
        return e   
    else:  
        if stream.pointer().type in ['SEMI']:  
            stream.match('SEMI')  
        return ('INTEGER', 0) 
```

```
def stmt(stream):  
    if stream.pointer().type in ['DECLARE']:  
        stream.match('DECLARE')  
        id_tok = stream.match('ID')  
        e = decl_suffix(stream)  
        if e[0] == 'FUNCTION':  
            (FUNCTION, args, body) = e  
            return ('FUNDECL', ('ID', id_tok.value), args, body)  
        else:  
            return ('VARDECL', ('ID', id_tok.value), e)  
    elif ...
```

Cuppa3 Frontend



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



```
(STMTLIST
| [
|   |(FUNDECL
|   |   |(ID add)
|   |   |(LIST
|   |   | [
|   |   |   |(ID a)
|   |   |   |(ID b))]
|   |   |(BLOCK
|   |   | (STMTLIST
|   |   | [
|   |   |   |(RETURN
|   |   |   |(PLUS
|   |   |   |(ID a)
|   |   |   |(ID b))))])
|   |(VARDECL
|   |   |(ID x)
|   |   |(INTEGER 0))] )
```

Cuppa3 Frontend



```
stmt : ID id_suffix

id_suffix : \( actual_args? \) ;?
           | = exp ;?
```

```
def stmt(stream):
    ...
    elif stream.pointer().type in ['ID']:
        id_tok = stream.match('ID')
        e = id_suffix(stream)
        if e[0] == 'LIST':
            return ('CALLSTMT', ('ID', id_tok.value), e)
        else:
            return ('ASSIGN', ('ID', id_tok.value), e)
    elif ...
```

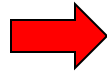
```
def id_suffix(stream):
    if stream.pointer().type in ['LPAREN']:
        stream.match('LPAREN')
        if stream.pointer().type in ['INTEGER', 'ID', 'LPAREN', 'MINUS', 'NOT']:
            args = actual_args(stream)
            stream.match('LPAREN')
            if stream.pointer().type in ['SEMI']:
                stream.match('SEMI')
            return args
    elif stream.pointer().type in ['ASSIGN']:
        stream.match('ASSIGN')
        e = exp(stream)
        if stream.pointer().type in ['SEMI']:
            stream.match('SEMI')
        return e
    else:
        raise SyntaxError("id_suffix: syntax error at {}".format(stream.pointer().value))
```

```
# actual_args : {INTEGER, ID, LPAREN, MINUS, NOT} exp ({COMMA} COMMA exp)*
def actual_args(stream):
    if stream.pointer().type in ['INTEGER', 'ID', 'LPAREN', 'MINUS', 'NOT']:
        e = exp(stream)
        ll = [e]
        while stream.pointer().type in ['COMMA']:
            stream.match('COMMA')
            e = exp(stream)
            ll.append(e)
        return ('LIST', ll)
    else:
        raise SyntaxError("actual_args: syntax error at {}".format(stream.pointer().value))
```


Cuppa3 Frontend



```
f(2,3);  
g = 5;
```



```
(STMTLIST  
 | [  
 | | (CALLSTMT  
 | | | (ID f)  
 | | | (LIST  
 | | | | [  
 | | | | | (INTEGER 2)  
 | | | | | (INTEGER 3) ] ) )  
 | | (ASSIGN  
 | | | (ID g)  
 | | | (INTEGER 5) ) ) ] )
```

Cuppa3 Frontend



```
primary : ID (\( actual_args? \))?
```

```
def primary(stream):
    ...
    elif stream.pointer().type in ['ID']:
        id_tk = stream.match('ID')
        if stream.pointer().type in ['LPAREN']:
            stream.match('LPAREN')
            if stream.pointer().type in ['INTEGER', 'ID', 'LPAREN', 'MINUS', 'NOT']:
                args = actual_args(stream)
            else:
                args = ('LIST', [])
            stream.match('RPAREN')
            return ('CALLEX', ('ID', id_tk.value), args)
        else:
            return ('ID', id_tk.value)
    elif ...
```

```
# actual_args : {INTEGER,ID,LPAREN,MINUS,NOT} exp ({COMMA} COMMA exp)*
def actual_args(stream):
    if stream.pointer().type in ['INTEGER', 'ID', 'LPAREN', 'MINUS', 'NOT']:
        e = exp(stream)
        ll = [e]
        while stream.pointer().type in ['COMMA']:
            stream.match('COMMA')
            e = exp(stream)
            ll.append(e)
        return ('LIST', ll)
    else:
        raise SyntaxError("actual_args: syntax error at {}".format(stream.pointer().value))
```

Cuppa3 Frontend



x = f(2,3) + y;



```
(STMTLIST
| [
| | (ASSIGN
| | | (ID x)
| | | (PLUS
| | | | (CALLEXP
| | | | | (ID f)
| | | | | (LIST
| | | | | | [
| | | | | | | (INTEGER 2)
| | | | | | | (INTEGER 3]))
| | | | (ID y))))]
```



Symbol Table

- The symbol table is extended so that we can manipulate scopes in order to implement *static scoping*

Symbol Table

cuppa3_symtab.py

```
class SymTab:

    def __init__(self):
        self.scoped_symtab = [{}]
```

def get_config(self):
we make a shallow copy of the symbol table
return list(self.scoped_symtab)

def set_config(self, c):
self.scoped_symtab = c

def push_scope(self):
...

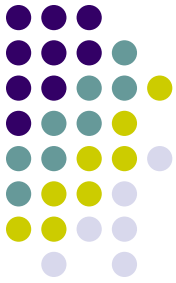
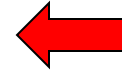
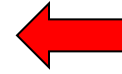
def pop_scope(self):
...

def declare_sym(self, sym, init):
...

def declare_fun(self, sym, init):
...

def lookup_sym(self, sym):
...

def update_sym(self, sym, val):
...



Interp Walker




Good News: the interpretation of the AST is the same as for Cuppa2 except for the nodes shown with the red arrow.

cuppa3_interp_walk.py

```
def walk(node):
    # node format: (TYPE, [child1[, child2[, ...]]])
    type = node[0]

    if type in dispatch:
        node_function = dispatch[type]
        return node_function(node)
    else:
        raise ValueError("walk: unknown tree node type: " + type)

# a dictionary to associate tree nodes with node functions
dispatch = {
    'STMTLIST': stmtlist,
    'NIL' : nil,
    'FUNDECL' : fundecl_stmt,
    'VARDECL' : vardecl_stmt,
    'ASSIGN' : assign_stmt,
    'GET' : get_stmt,
    'PUT' : put_stmt,
    'CALLSTMT' : call_stmt,
    'RETURN' : return_stmt,
    'WHILE' : while_stmt,
    'IF' : if_stmt,
    'BLOCK' : block_stmt,
    'INTEGER' : integer_exp,
    'ID' : id_exp,
    'CALLEXP' : call_exp,
    'PAREN' : paren_exp,
    'PLUS' : plus_exp,
    'MINUS' : minus_exp,
    'MUL' : mul_exp,
    'DIV' : div_exp,
    'EQ' : eq_exp,
    'LE' : le_exp,
    'UMINUS' : uminus_exp,
    'NOT' : not_exp
}
```





Interpreting Declarations

- We now have two types of values that we need to store in the symbol table
 - Integer values
 - Function values
- We tag the values that we store in the symbol table with appropriate type tags
 - Traditionally this is called a 'symbol table record'
 - For us it is just a tuple of type tag and value



Interpreting Declarations

```
def vardecl_stmt(node):  
  
    (VARDECL, (ID, name), init_val) = node  
  
    value = walk(init_val)  
    symtab.declare(name, ('INTEGER', value))  
    return None
```



Function context needed
for static scoping

```
def fundecl_stmt(node):  
  
    (FUNDECL, (ID, name), arglist, body) = node  
  
    context = symtab.get_config()  
    funval = ('FUNVAL', arglist, body, context)  
    symtab.declare(name, funval)  
    return None
```






Interpreting Assignments

- The fact that we are binding tuples into the symbol table affects assignment statements
- We have to bind tuples into the symbol table for assigned values.

```
def assign_stmt(node):  
  
    (ASSIGN, (ID, name), exp) = node  
  
    value = walk(exp)  
    symtab.update_sym(name, ('INTEGER', value))  
  
    return None
```

A red arrow pointing upwards from below the code block to the string 'INTEGER' in the line `symtab.update_sym(name, ('INTEGER', value))`.

Interpreting Identifier Expressions



- Variables that appear in expressions return values
- Before we can return a value, we need to unpack the structure bound into the symbol table

```
def id_exp(node):  
  
    (ID, name) = node  
  
    val = symtab.lookup_sym(name)  
  
    if val[0] != 'INTEGER':  
        raise ValueError("{} is not an integer".format(name))  
  
    return val[1]
```



Interpreting Function Calls

- The difference between call statements and call expressions:
 - Call statements – return value of a function is ignored
 - Call expressions – function has to provide a return value

Note: the return value of functions called as statement is ignored.
Consider:

```
declare f () {  
  put(1001);  
  return 1001;  
}
```

```
f();
```

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

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

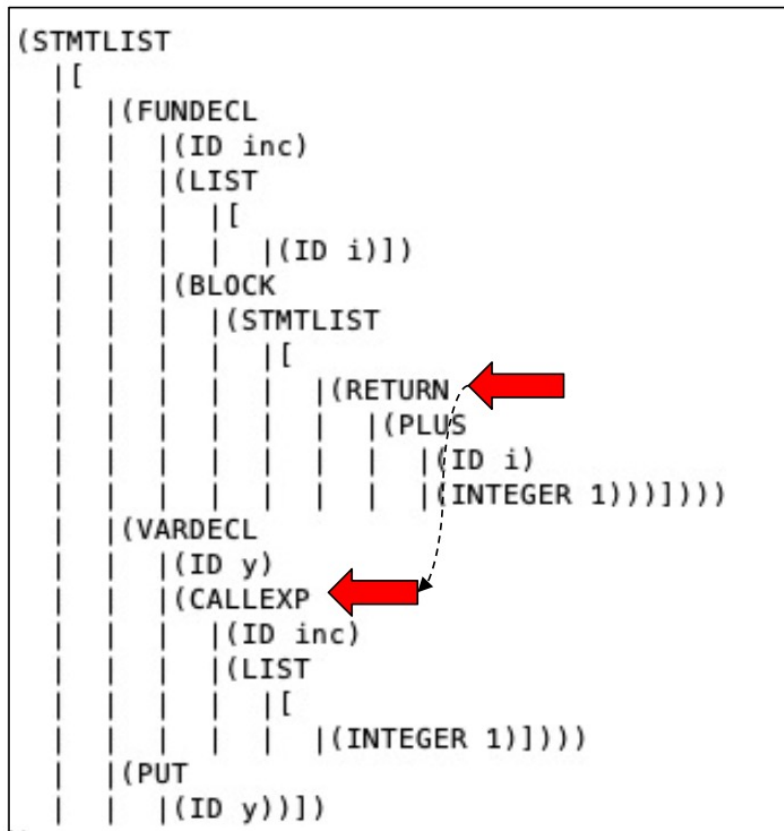


Interpreting Function Calls

- How do we get function return values to the call site?
 - We *throw* them!

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

declare y = inc(1);
put y;
```



Interpreting Function Calls



```
def call_stmt(node):  
  
    (CALLSTMT, (ID, name), actual_args) = node  
  
    handle_call(name, actual_args)  
  
    return None
```

```
def return_stmt(node):  
  
    (RETURN, exp) = node  
  
    value = walk(exp)  
    raise ReturnValue(value)
```

```
def call_exp(node):  
  
    (CALLEXP, (ID, name), actual_args) = node  
  
    return_value = handle_call(name, actual_args)  
  
    if return_value is None:  
        raise ValueError("No return value from function {}".format(name))  
  
    return return_value
```

```
class ReturnValue(Exception):  
  
    def __init__(self, value):  
        self.value = value  
  
    def __str__(self):  
        return(repr(self.value))
```

```
def nil(node):  
  
    (NIL,) = node  
  
    # do nothing!  
    return None
```

Interpretin

'handle_call' our function
call work horse

```
def handle_call(name, actual_arglist):
    """
    handle calls for both call-statements and call-expressions.
    """

    val = symtab.lookup_sym(name)

    if val[0] != 'FUNVAL':
        raise ValueError("{} is not a function".format(name))

    # unpack the funval tuple
    (FUNVAL, formal_arglist, body, context) = val

    # set up the environment for static scoping and then execute the function
    actual_val_args = eval_actual_args(actual_arglist)
    save_symtab = symtab.get_config()
    symtab.set_config(context)
    symtab.push_scope()
    declare_formal_args(formal_arglist, actual_val_args)

    # execute function
    return_value = None
    try:
        walk(body)
    except ReturnValue as val:
        return_value = val.value

    # NOTE: popping the function scope is not necessary because we
    # are restoring the original symtab configuration
    symtab.set_config(save_symtab)

    return return_value
```



Interpreting Function Calls

```
def eval_actual_args(args):  
    '''  
    Walk the list of actual arguments, evaluate them, and  
    return a list with the evaluated actual values  
    '''  
    (LIST, ll) = args  
  
    outlist = []  
    for e in ll:  
        v = walk(e)  
        outlist.append(('INTEGER', v))  
  
    return ('LIST', outlist)
```

```
def declare_formal_args(formal_args, actual_val_args):  
    '''  
    Walk the formal argument list and declare the identifiers on that  
    list using the corresponding actual args as initial values.  
    NOTE: this is where we implement by-value argument passing  
    '''  
    (LIST, fl) = formal_args  
    (LIST, avl) = actual_val_args  
  
    if len(fl) != len(avl):  
        raise ValueError("actual and formal argument lists do not match")  
  
    for ((ID, f), v) in zip(fl, avl):  
        symtab.declare(f, v)
```

Driver Function



```
def interp(input_stream, dump=False, exceptions=False):
    try:
        symtab.initialize()
        ast = parse(input_stream)
        if dump:
            dumpast(ast)
        else:
            walk(ast)
    except Exception as e:
        if exceptions:
            raise e # rethrow for visibility
        else:
            print("error: "+str(e))
    return None
```




Testing the Interpreter

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

// ask the user for input
declare v;
get v;
put fact(v);
```

```
$ python3 cuppa3_interp.py fact.txt
Value for v? 3
6
$
```

Assignment



- Assignment #5 – see BrightSpace