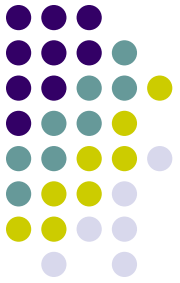




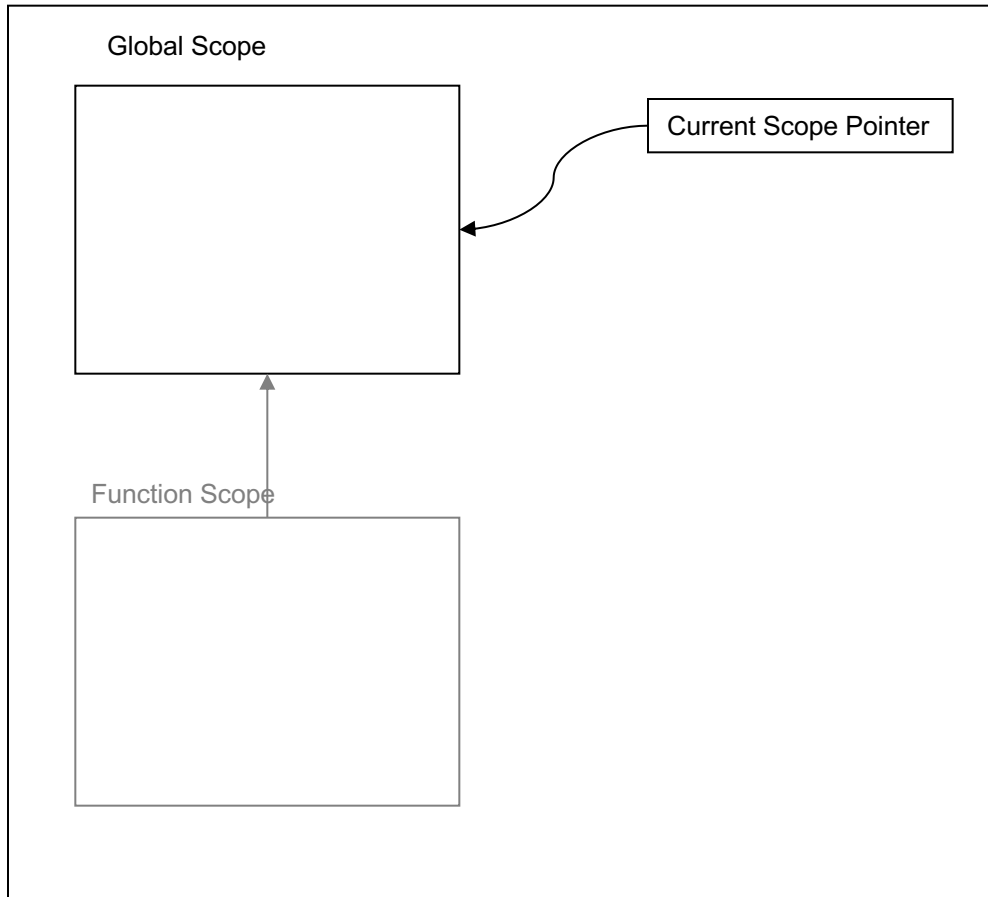
Interpreter Implementation

- 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



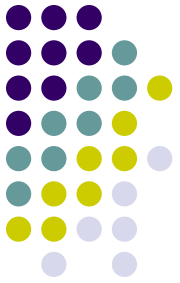
Symbol Table



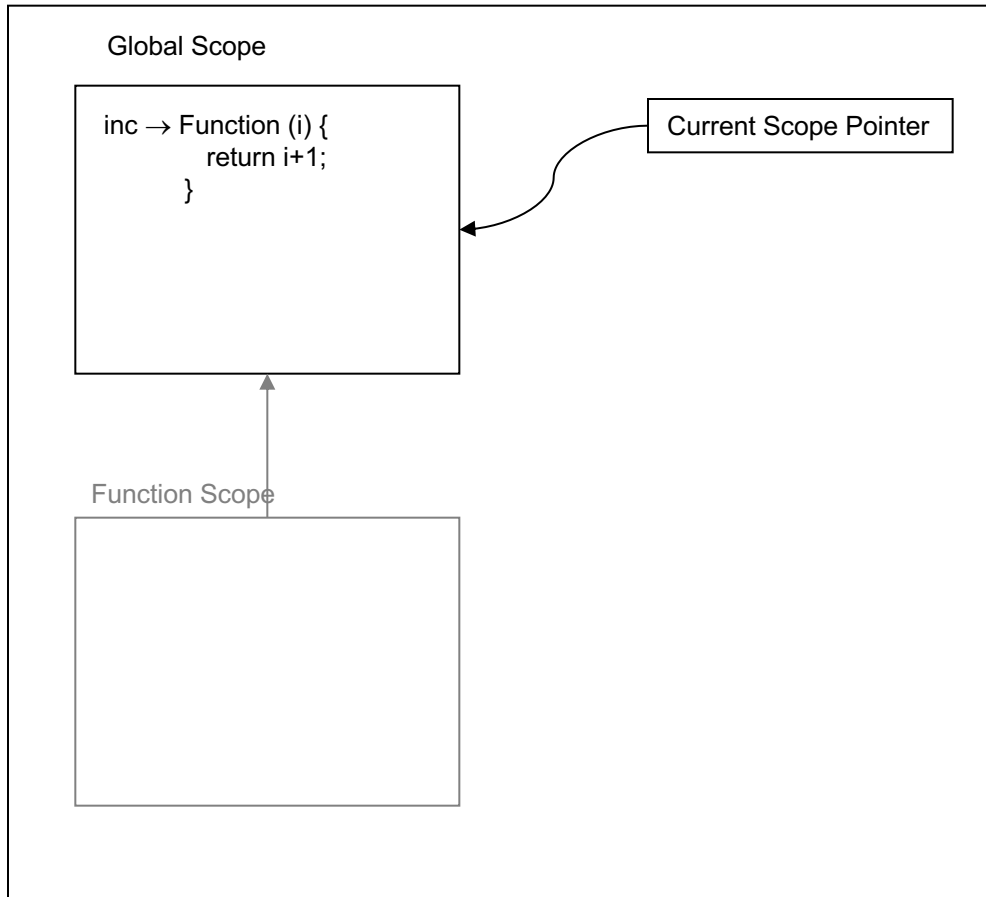
```
declare inc(i) {  
    return i+1;  
}
```

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

Interpreting Functions



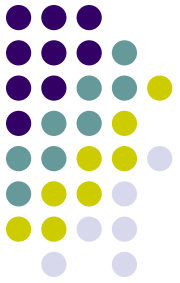
Symbol Table



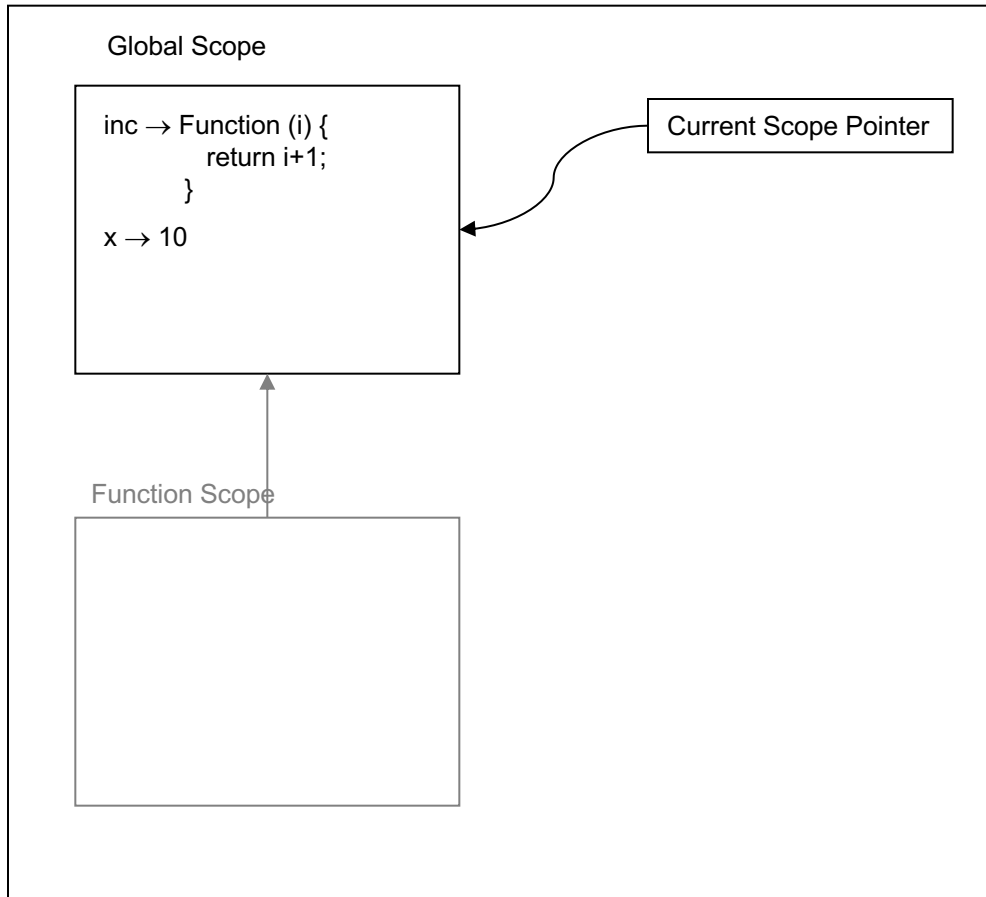
→ `declare inc(i) {
 return i+1;
}`

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

Interpreting Functions

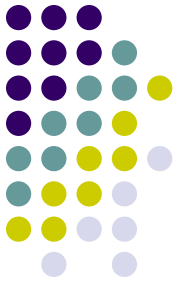


Symbol Table


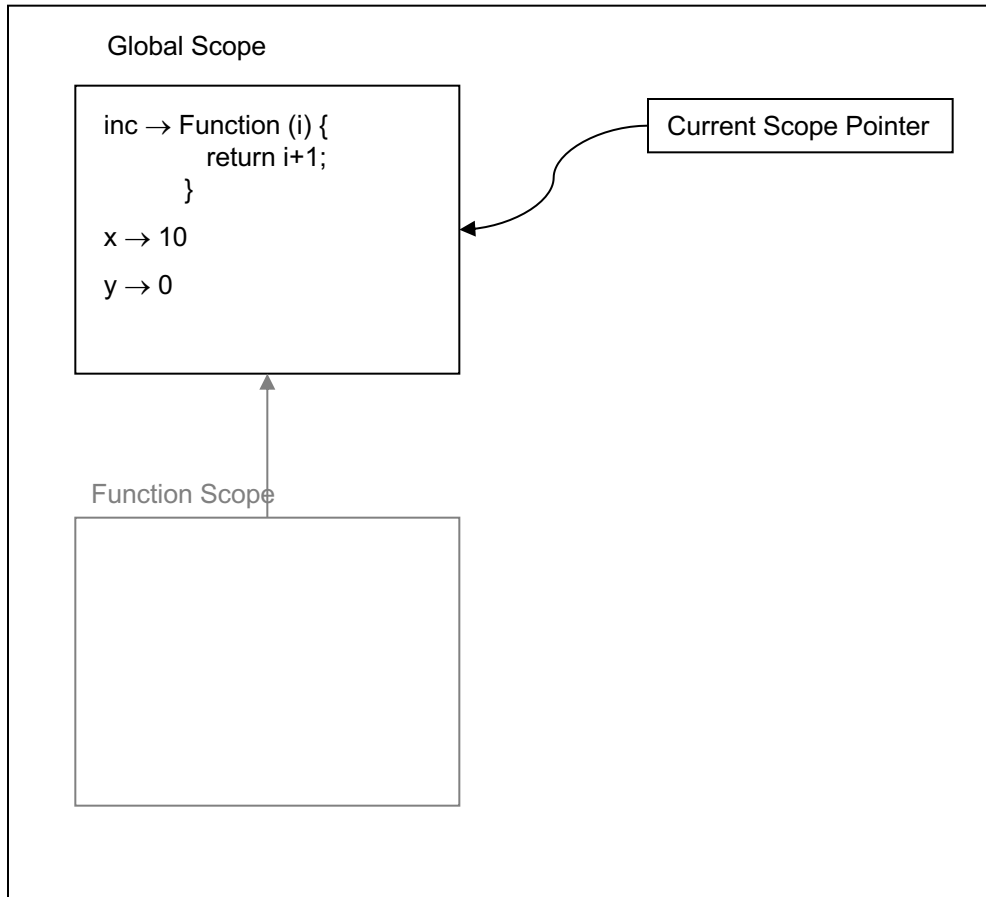


```
declare inc(i) {  
    return i+1;  
}  
  
declare x = 10;  
declare y;  
y = inc(x);  
put y;
```

Interpreting Functions



Symbol Table

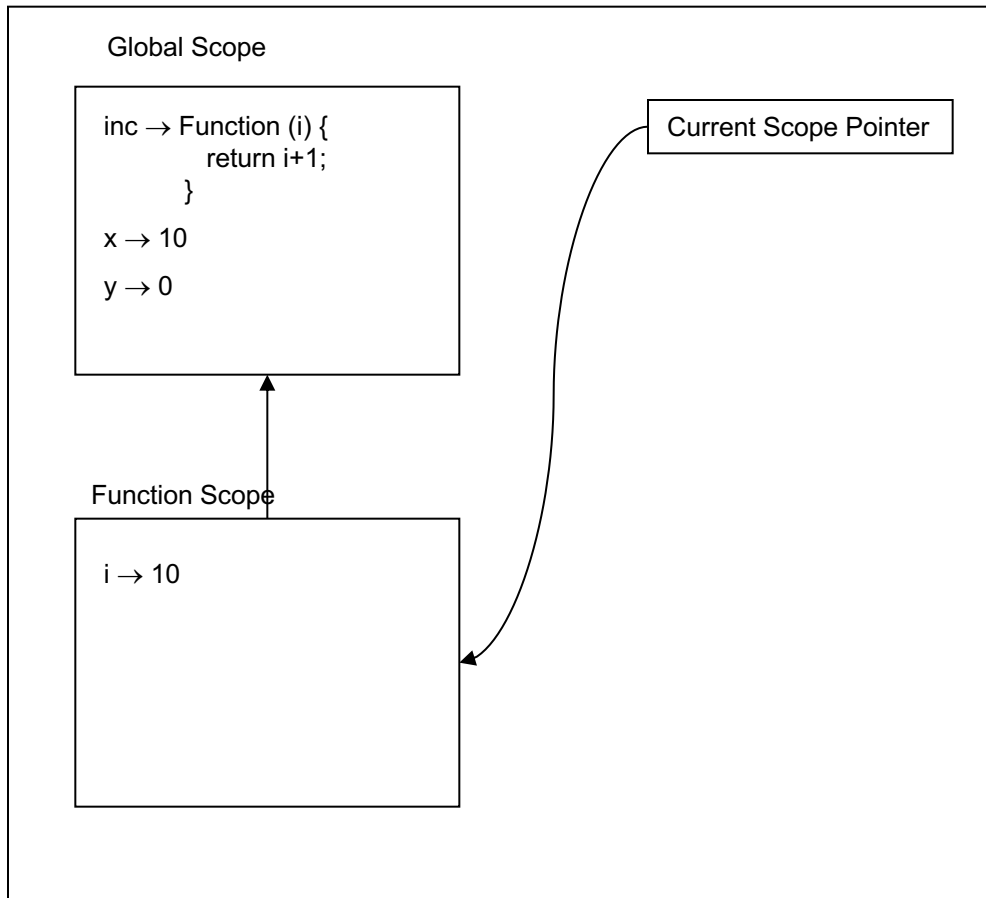


```
declare inc(i) {  
    return i+1;  
}  
  
declare x = 10;  
declare y;  
y = inc(x);  
put y;
```

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;
}

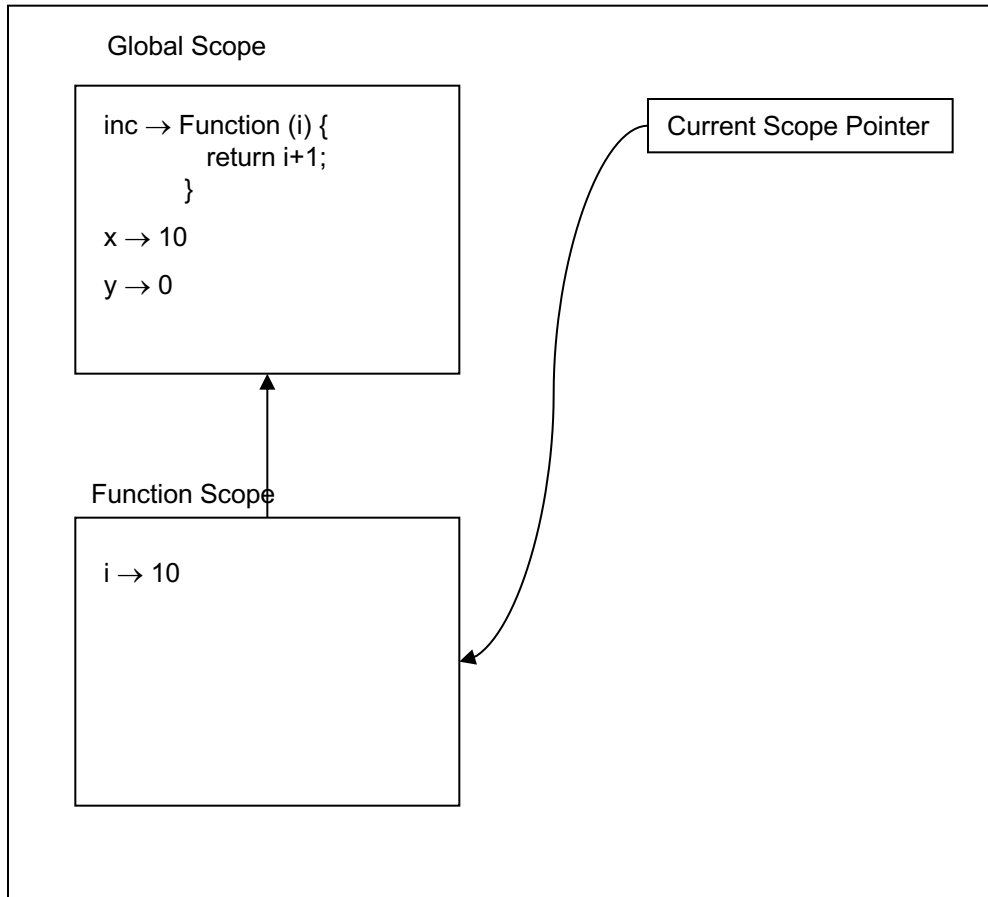
Setup the function call:

- 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;
```



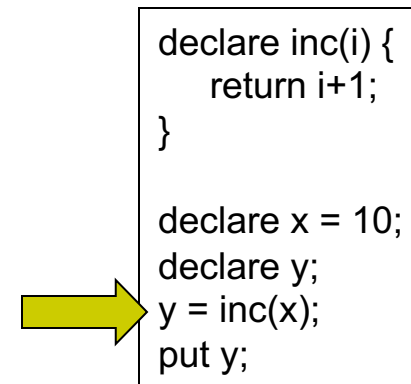
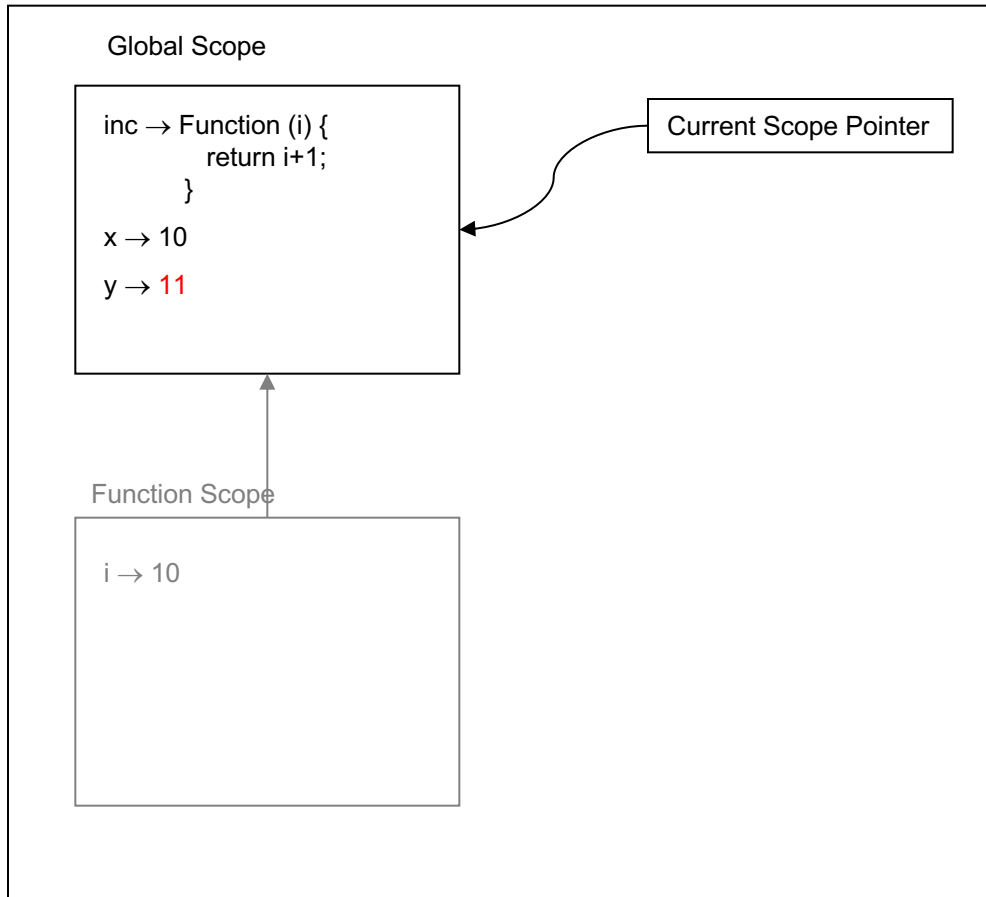
```
Function (i) {  
    return i+1;  
}
```

Execute the called function and compute return value.

Interpreting Functions

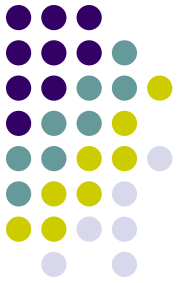


Symbol Table

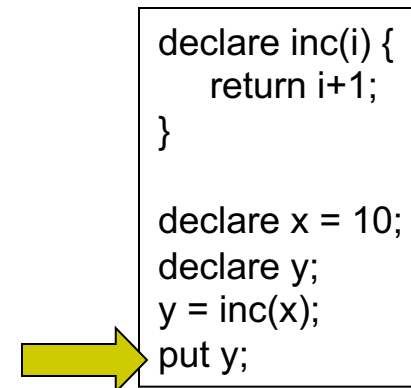
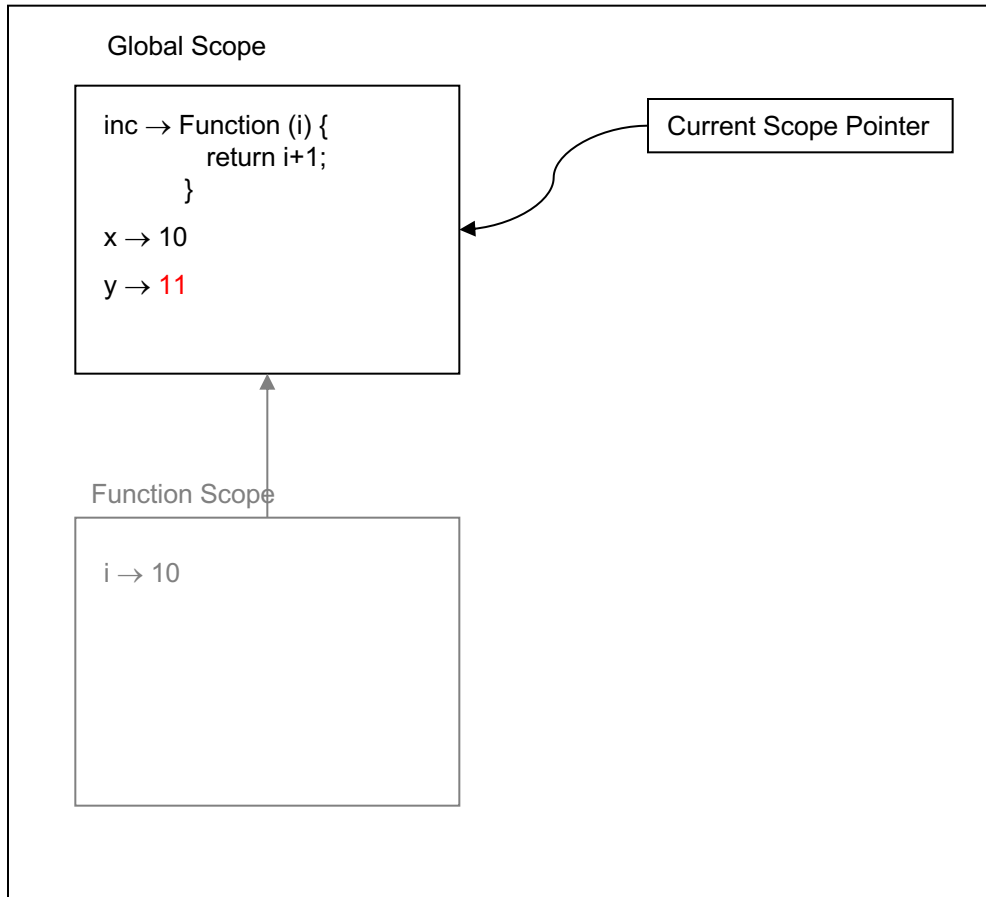


- Exit the called function:
- pop the function scope
 - store the return value in y

Interpreting Functions

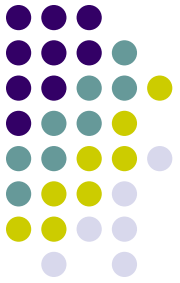


Symbol Table

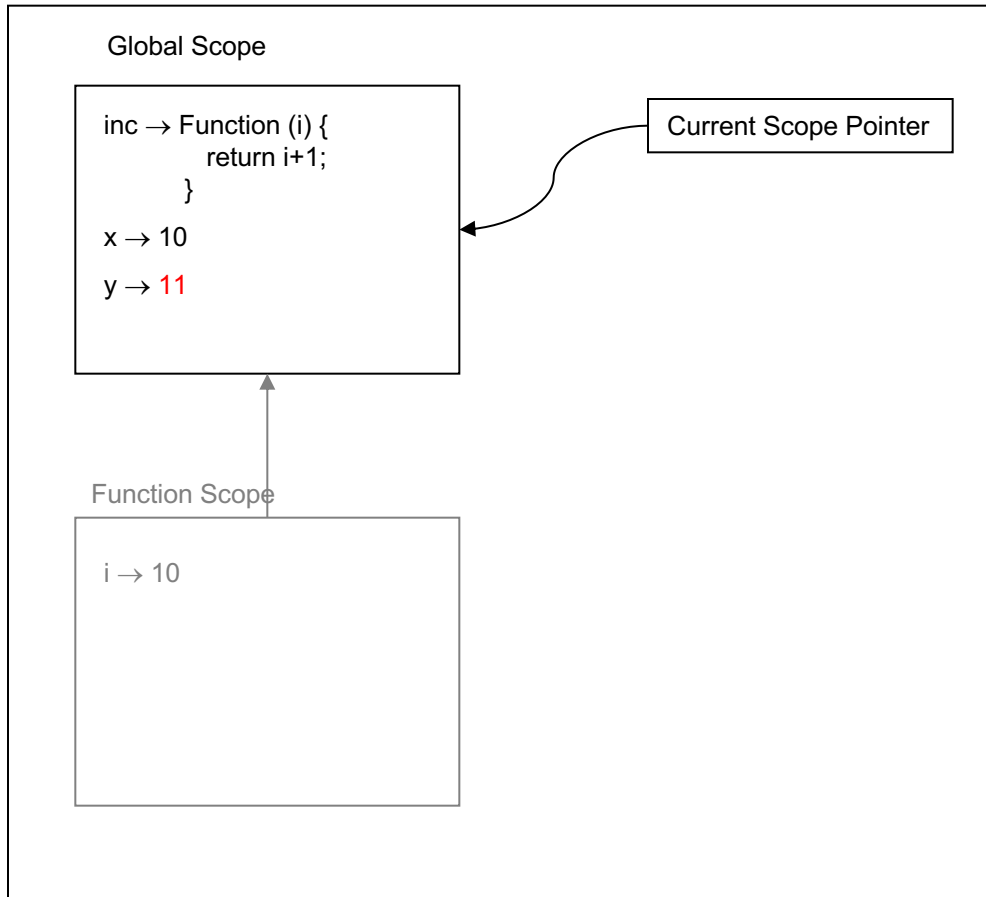


Execute the put statement \Rightarrow 11

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

```
def p_stmt(p):
    '''
    stmt : DECLARE ID '(' opt_formal_args ')' stmt
          | DECLARE ID opt_init opt_semi
          | ID '=' exp opt_semi
          | GET ID opt_semi
          | PUT exp opt_semi
          | ID '(' opt_actual_args ')' opt_semi
          | RETURN opt_exp opt_semi
          | WHILE '(' exp ')' stmt
          | IF '(' exp ')' stmt opt_else
          | '{' stmt_list '}'
    '''
    if p[1] == 'declare' and p[3] == '(':
        p[0] = ('fundekl', p[2], p[4], p[6])
    elif p[1] == 'declare':
        p[0] = ('declare', p[2], p[3])
    elif is_ID(p[1]) and p[2] == '=':
        p[0] = ('assign', p[1], p[3])
    elif p[1] == 'get':
        p[0] = ('get', p[2])
    elif p[1] == 'put':
        p[0] = ('put', p[2])
    elif is_ID(p[1]) and p[2] == '(':
        p[0] = ('callstmt', p[1], p[3])
    elif p[1] == 'return':
        p[0] = ('return', p[2])
    elif p[1] == 'while':
        p[0] = ('while', p[3], p[5])
    elif p[1] == 'if':
        p[0] = ('if', p[3], p[5], p[6])
    elif p[1] == '{':
        p[0] = ('block', p[2])
    else:
        raise ValueError("unexpected symbol {}".format(p[1]))
```

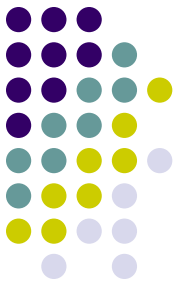
```
def p_opt_formal_args(p):
    '''
    opt_formal_args : formal_args
                    | empty
    '''
    p[0] = p[1]

#####
def p_formal_args(p):
    '''
    formal_args : ID ',' formal_args
                | ID
    '''
    if (len(p) == 4):
        p[0] = ('seq', ('id', p[1]), p[3])
    elif (len(p) == 2):
        p[0] = ('seq', ('id', p[1]), ('nil',))
```

```
def p_opt_actual_args(p):
    '''
    opt_actual_args : actual_args
                    | empty
    '''
    p[0] = p[1]

#####
def p_actual_args(p):
    '''
    actual_args : exp ',' actual_args
                | exp
    '''
    if (len(p) == 4):
        p[0] = ('seq', p[1], p[3])
    elif (len(p) == 2):
        p[0] = ('seq', p[1], ('nil',))
```

```
def p_call_exp(p):
    '''
    exp : ID '(' opt_actual_args ')'
    '''
    p[0] = ('callexp', p[1], p[3])
```





Symbol Table

- The symbol table is extended to store two different kinds of objects:
 - Scalars
 - Functions
- It is also 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):  
        ...
```

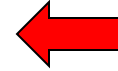
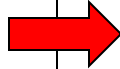
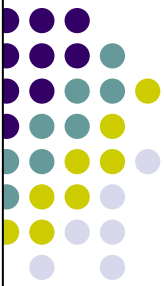
```
    def declare_sym(self, sym, init):  
        # declare the scalar in the current scope: dict @ position 0  
  
        # first we need to check whether the symbol was already declared  
        # at this scope  
        if sym in self.scoped_symtab[Curr_Scope]:  
            raise ValueError("symbol {} already declared".format(sym))  
  
        # enter the symbol in the current scope  
        scope_dict = self.scoped_symtab[Curr_Scope]  
        scope_dict[sym] = ('scalar', init)
```

```
    def declare_fun(self, sym, init):  
        # declare a function in the current scope: dict @ position 0  
  
        # first we need to check whether the symbol was already declared  
        # at this scope  
        if sym in self.scoped_symtab[Curr_Scope]:  
            raise ValueError("symbol {} already declared".format(sym))
```

```
        # enter the function in the current scope  
        scope_dict = self.scoped_symtab[Curr_Scope]  
        scope_dict[sym] = ('function', 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_dict:
        node_function = dispatch_dict[type]
        return node_function(node)
    else:
        raise ValueError("walk: unknown tree node type: " + type)

# a dictionary to associate tree nodes with node functions
dispatch_dict = {
    'seq'      : seq,
    'nil'      : nil,
    'fundecl'  : fundecl_stmt,
    'declare'  : declare_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_exp,
    '-'       : minus_exp,
    '*'       : times_exp,
    '/'       : divide_exp,
    '=='      : eq_exp,
    '<='      : le_exp,
    'uminus'  : uminus_exp,
    'not'     : not_exp
}
```



Interp Walk

- 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;
```



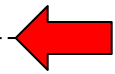

Interp Walk

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

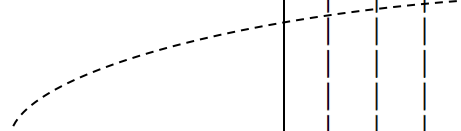
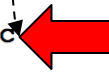
```
declare inc(i)
{
    return i+1;
}

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

```
(seq
 | (fundecl inc
   | (seq
     | (id i)
     | (nil)))
 | (block
   | (seq
     | (return
       | (+
         | (id i)
         | (integer 1)))
     | (nil))))
| (nil))
```



```
(seq
 | (declare y
   | (callexp inc
     | (seq
       | (integer 1)
       | (nil))))
 | (seq
   | (put
     | (id y))
   | (nil)))
```



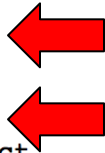


Interp Walk

- Throwing the return value also solves the problem of terminating a deeply recursive computation on the AST!

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

```
(seq
 | (fundecl fact
 | (seq
 | (id x)
 | (nil))
 | (block
 | (seq
 | (if
 | (<=
 | (id x)
 | (integer 1))
 | (return
 | (integer 1))
 | (return
 | (*
 | (callexp fact
 | (seq
 | (-
 | (id x)
 | (integer 1))
 | (nil)))
 | (id x))))
 | (nil))))
 | (nil))
```



Interp Walk



```
def fundecl_stmt(node):

    try: # try the fundecl pattern without arglist
        (FUNDECL, name, (NIL,), body) = node
        assert_match(FUNDECL, 'fundecl')
        assert_match(NIL, 'nil')

    except ValueError: # try fundecl with arglist
        (FUNDECL, name, arglist, body) = node
        assert_match(FUNDECL, 'fundecl')

        context = state.symbol_table.get_config()
        funval = ('funval', arglist, body, context)
        state.symbol_table.declare_fun(name, funval)

    else: # fundecl pattern matched
        # no arglist is present
        context = state.symbol_table.get_config()
        funval = ('funval', ('nil',), body, context)
        state.symbol_table.declare_fun(name, funval)
```

```
def call_stmt(node):

    (CALLSTMT, name, actual_args) = node
    assert_match(CALLSTMT, 'callstmt')

    handle_call(name, actual_args)
```

```
def call_exp(node):
    # call_exp works just like call_stmt with the exception
    # that we have to pass back a return value

    (CALLEXP, name, args) = node
    assert_match(CALLEXP, 'callexp')

    return_value = handle_call(name, args)

    if return_value is None:
        raise ValueError("No return value from function {}".format(name))

    return return_value
```

```
def return_stmt(node):
    # if a return value exists the return stmt will record it
    # in the state object

    try: # try return without exp
        (RETURN, (NIL,)) = node
        assert_match(RETURN, 'return')
        assert_match(NIL, 'nil')

    except ValueError: # return with exp
        (RETURN, exp) = node
        assert_match(RETURN, 'return')

        value = walk(exp)
        raise ReturnValue(value)

    else: # return without exp
        raise ReturnValue(None)
```

Interp Walk

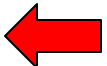


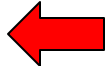
```
class ReturnValue(Exception):  
  
    def __init__(self, value):  
        self.value = value  
  
    def __str__(self):  
        return(repr(self.value))
```

```
def handle_call(name, actual_arglist):  
  
    (type, val) = state.symbol_table.lookup_sym(name)  
  
    if type != 'function':  
        raise ValueError("{} is not a function".format(name))  
  
    # unpack the funval tuple  
    (FUNVAL, formal_arglist, body, context) = val  
  
    if len_seq(formal_arglist) != len_seq(actual_arglist):  
        raise ValueError("function {} expects {} arguments".format(sym, len_seq(formal_arglist)))  
  
    # set up the environment for static scoping and then execute the function  
    actual_val_args = eval_actual_args(actual_arglist) # evaluate actuals in current symtab  
    save_symtab = state.symbol_table.get_config()      # save current symtab  
    state.symbol_table.set_config(context)             # make function context current symtab  
    state.symbol_table.push_scope()                   # push new function scope  
    declare_formal_args(formal_arglist, actual_val_args) # declare formals in function scope  
  
    return_value = None  
    try:  
        walk(body)                                     # execute the function  
    except ReturnValue as val:  
        return_value = val.value  
  
    state.symbol_table.pop_scope()                     # pop function scope  
    state.symbol_table.set_config(save_symtab)         # restore original symtab config  
  
    return return_value
```

Interp Walk



```
def eval_actual_args(args):  
  
    if args[0] == 'nil':  
        return ('nil',)  
  
    elif args[0] == 'seq':  
        # unpack the seq node  
        (SEQ, p1, p2) = args  
  
        val = walk(p1)   
  
        return ('seq', val, eval_actual_args(p2))  
  
    else:  
        raise ValueError("unknown node type: {}".format(args[0]))
```

```
def declare_formal_args(formal_args, actual_val_args):  
  
    if len_seq(actual_val_args) != len_seq(formal_args):  
        raise ValueError("actual and formal argument lists do not match")  
  
    if formal_args[0] == 'nil':  
        return  
  
    # unpack the args  
    (SEQ, (ID, sym), p1) = formal_args  
    (SEQ, val, p2) = actual_val_args  
  
    # declare the variable   
    state.symbol_table.declare_sym(sym, val)  
  
    declare_formal_args(p1, p2)
```

Driver Function



```
def interp(input_stream):  
  
    # initialize the state object  
    state.initialize()  
  
    # build the AST  
    parser.parse(input_stream, lexer=lexer)  
  
    # walk the AST  
    #dump_AST(state.AST)  
    walk(state.AST)
```

Testing the Interpreter



```
add = \  
'''  
declare add(a,b)  
{  
    return a+b;  
}  
  
declare x = add(3,2);  
put x;  
'''
```

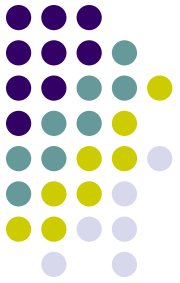
```
In [5]: interp(add)  
  
> 5
```

```
factrec = \  
'''  
// 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);  
'''
```

```
In [9]: interp(factrec)  
  
Value for v? 3  
> 6
```

Assignment

- Final project proposal



Assignment

- Assignment #6

