CptS 355- Programming Language Design

Scope and Scoping

Instructor: Sakire Arslan Ay

WASHINGTON STATE UNIVERSITY

World Class. Face to Face.

Outline

- Storage Management in Programming Languages
- Scope
- Referencing environments
- Lifetime
- Static scope rules
- Dynamic scope rules

Some Terminology:

- Run-time:
  - refers to the time when an application actually executes.
- Compile-time:
  - everything before run-time, that is, compilation, linking, and loading.
- For a program, its static structure is the structure of the source program, how it is organized.
- The dynamic structure is the structure that evolves during run-time.

Storage Management

- How storage is managed in different programming languages and for different kinds of data?
- Storage is typically divided into 3 areas:
  - Static area
  - Stack area
  - Heap
- Stack and heap is used for dynamic allocation

Static Allocation

- Static storage allocation is appropriate when the storage requirements are known at compile time.

- Static allocation for
  - Machine language translation of the source code
  - Global variables (in C)
  - Static variables
  - Constants (including strings, sets, etc.)

- With ***static*** storage, the location of every variable is fixed, allocated and known at compile-time. In principle, every variable has a fixed constant machine address.

Stack-based Allocation

- Stack-based storage allocation is appropriate when the storage requirements are not known at compile time, but the requests obey a last-in, first-out discipline.

- Stack is an efficient way to provide the storage needed for function calls (execution of program blocks).

Stack-based Allocation

- When a function is called, all storage needed for the function is allocated on the stack in a section called the activation record (also called the stack frame).
- Activation record includes:
  - return address,
  - the return value (possibly a pointer),
  - actual parameters passed to the function,
  - any variables declared within the function, including:
    - static arrays (C and C++ only), static structs and classes (C++ only).
  - Also holds locations for temporary values, and pointers to other parts of the stack (to facilitate access and deallocation).
- On return from the function, the storage on the stack is deallocated.

Stack-based Allocation

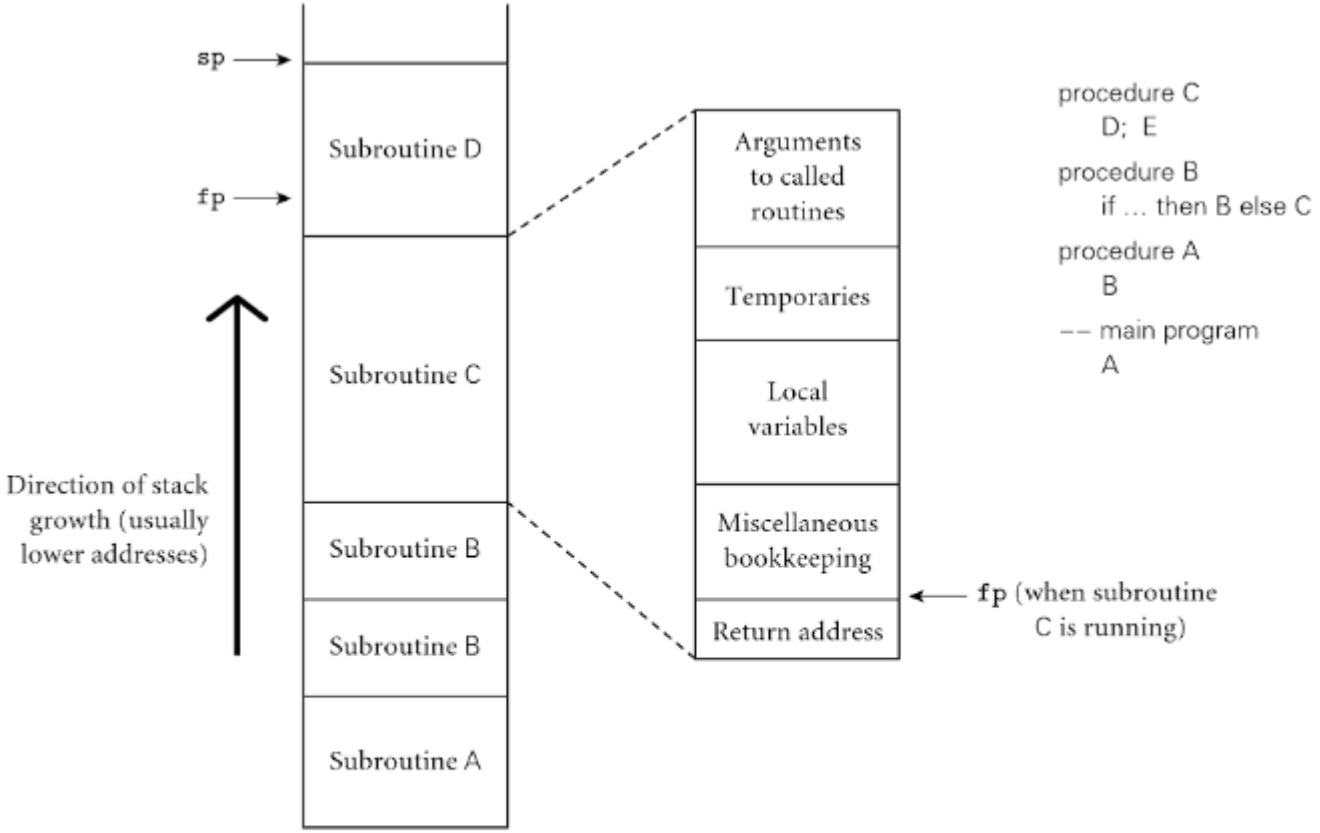


Figure 3.1 Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

Stack-based Allocation

- "Why not use static storage?":
  - The static method is simple to implement and efficient in execution, but it is more wasteful of storage than other methods.
  - One can't implement recursion statically.
  - One can't have dynamic data structures, that is, the structure of the data can't be decided on and created at run-time.

Heap-based Allocation

- Allocation & deallocation can be done in random order.
- Space management issues:
  - Internal fragmentation
    - A block that is larger than required is allocated
    - Extra space is wasted
  - External fragmentation
    - Although there is sufficient space, the unused space is scattered and no one piece is large enough to satisfy a request

Scope and Referencing Environment

Scope:

- Definition: The scope of a name binding (declaration) is the part of the computer program where the binding is visible.

```
1:  {int x = 3;
2:     {int x=5;
3:          {int y=x;}
4:          print(x);
5:     }
6:     print(x);
7:   }
```

```
1:  {int x = 3;
2:     {int x=5;
3:          {int y=x;}
4:          print(x);
5:     }
6:     print(x);
7:   }
```

The inner declaration of x shadows (hides) the outer one

Scope and Referencing Environment

Referencing environment:

- Definition: The referencing environment of a program location is the collection of the declarations whose variables are visible at that location.

```
1:  {int x = 3;
2:     {int x=5;
3:          {int y=x; }    x=5, y=5
4:          print(x);
5:     }
6:     print(x);
7:   }
```

Scope and Referencing Environment

Lifetime:

- Definition: The lifetime of a variable is from when it is allocated to when it is no longer accessible (expressed as time)

```
1:  {int x = 3;
2:     {int x=5;
3:          {int y=x; }
4:          print(x);
5:     }
6:     print(x);
7:   }
```

lifetime of x

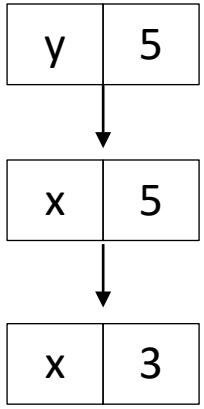
The inner declaration of x shadows (hides) the outer one

Note the difference between scope and lifetime!

Activation Record

- Each block's variables are held in a separate activation record (AR) (also called *stack frame*) and the activation records live in a stack

```
1:  {int x = 3;
2:     {int x=5;
3:          {int y=x;}
4:          print(x);
5:     }
6:     print(x);
7:  }
```

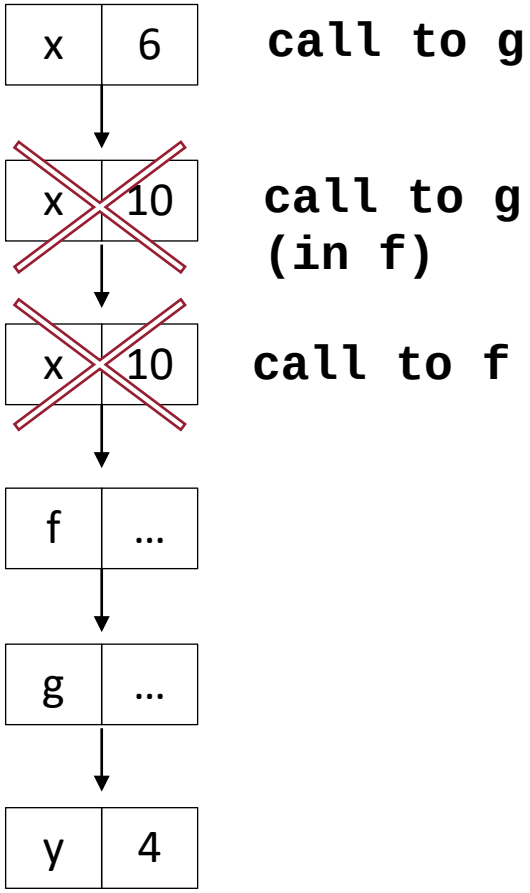


A *block* is the body of a module, class, subroutine, or structured control-flow statement. (In C family languages blocks are determined by {…} braces.)

Activation Records

- **Haskell Example-1:**

```
y = 4
g x = x+y
f x = g x
f 10
g 6
```



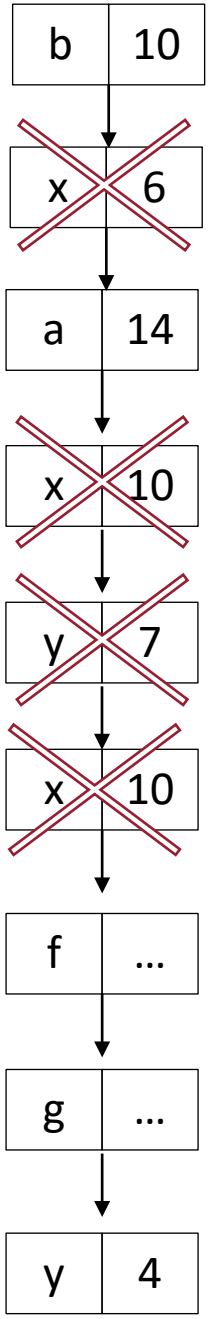
In Haskell, three actions that occur in a program can cause us to create an activation record

- Function definition
- Variable definition
- Function application (i.e., function call)

Activation Records

- **Haskell Example-2:**

```
y=4
g x = x+y
f x = let
          val y = 7
      in
          g x
a = f 10
b = g 6
```



call to g

call to g (in f)

f's let block

call to f

Static Scoping and Dynamic Scoping

Static Scoping:

Definition:

The identifier refers to the declaration in the closest and closing block.

Dynamic Scoping:

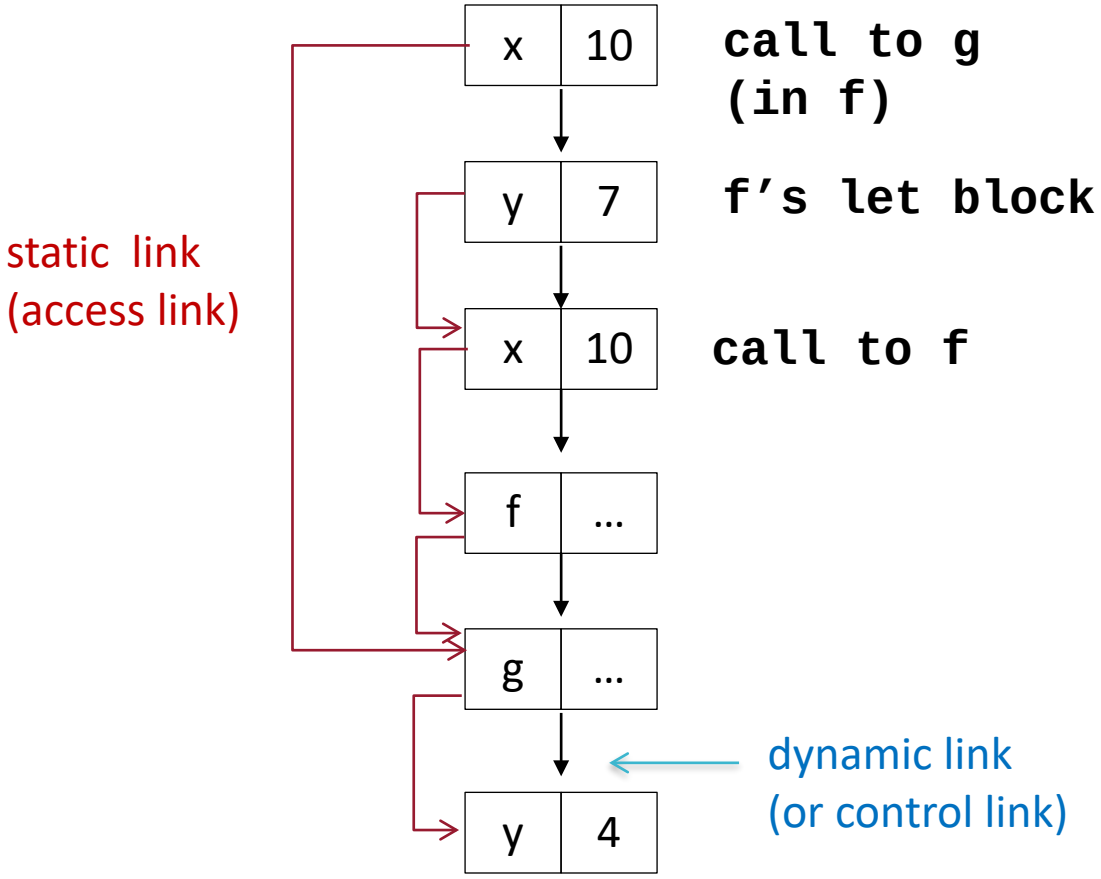
Definition

The identifier refers to the most recent (in time) still-live declaration

Static Scoping

- **Haskell Example:**

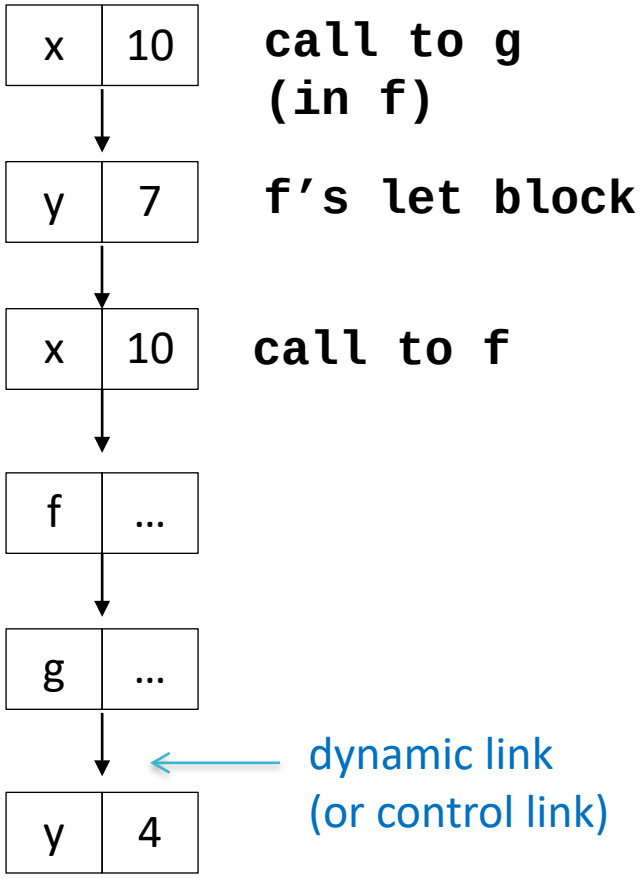
```
y=4
g x = x+y
f x = let
          val y = 7
      in
          g x
a = f 10
b = g 6
```



Dynamic Scoping

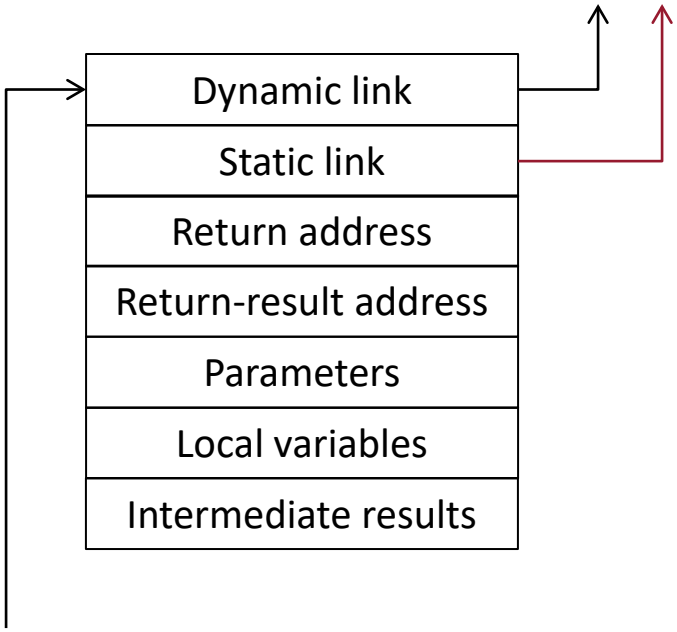
- **Haskell Example:**

```
y=4
g x = x+y
f x = let
          val y = 7
      in
          g x
a = f 10
b = g 6
```

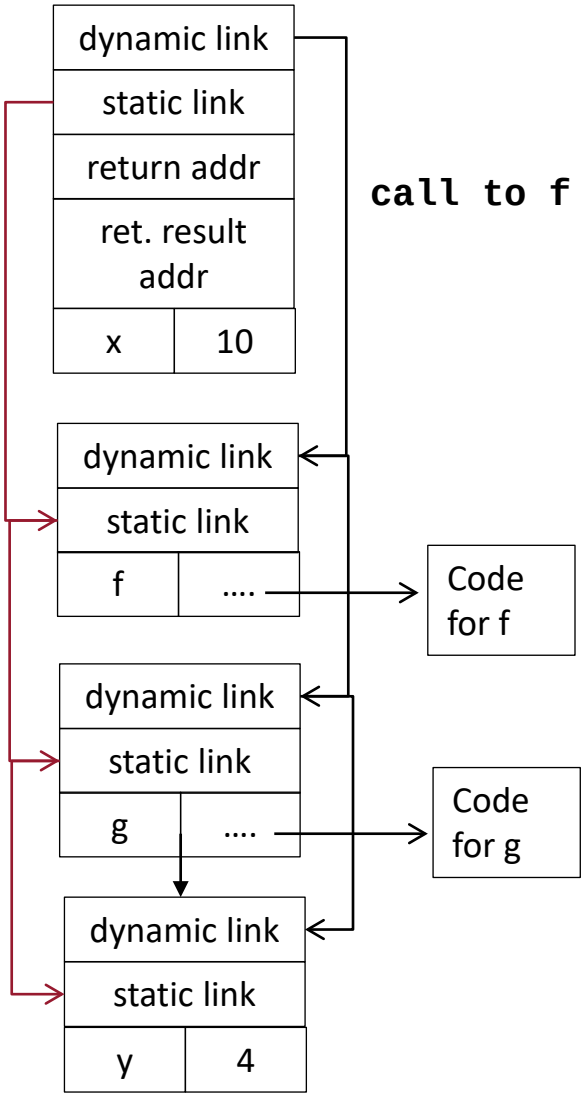


Activation Records - revisited

- Activation record with access link for functions call with static scope




```
y=4
g x = x+y
f x = let
          val y = 7
      in
          g x
a = f 10
b = g 6
```



Scoping Example -2a

```
int m, n;
m = 50;
n = 100;
procedure egg();
begin
        print("in egg-- n = ", n);
end;

procedure chicken (n: integer);
begin
        print("in chicken -- m = ", m);
        print("in chicken -- n = ", n);
        egg();
end;
print("in main program -- n = ", n);
chicken(1);
egg();
```

The output using static scope rules:

```
in main program -- n =
in chicken -- m =
in chicken -- n =
in egg -- n =
/* note that here egg is called from chicken*/
in egg -- n =
/* here egg is called from the main program */
```

The output using dynamic scope rules:

```
in main program -- n =
in chicken -- m =
in chicken -- n =
in egg -- n =
/*NOTE DIFFERENCE -- here egg is called from chicken
in egg-- n =
/* here egg is called from the main program*/
```

Scoping Example -2b- Nested Functions

```
int m, n;
m = 50;
n = 100;
procedure chicken (n: integer);
begin
        procedure egg();
        begin
              print("in egg-- n = ", n);
        end;

        print("in chicken- m = ", m);
        print("in chicken- n = ", n);
        egg();
end;
print("in main program -- n = ", n);
chicken(1);
/* can't call egg from the main program any more */
```

- **Static scope**: use environment where function is defined
- **Dynamic scope**: use environment where function is called

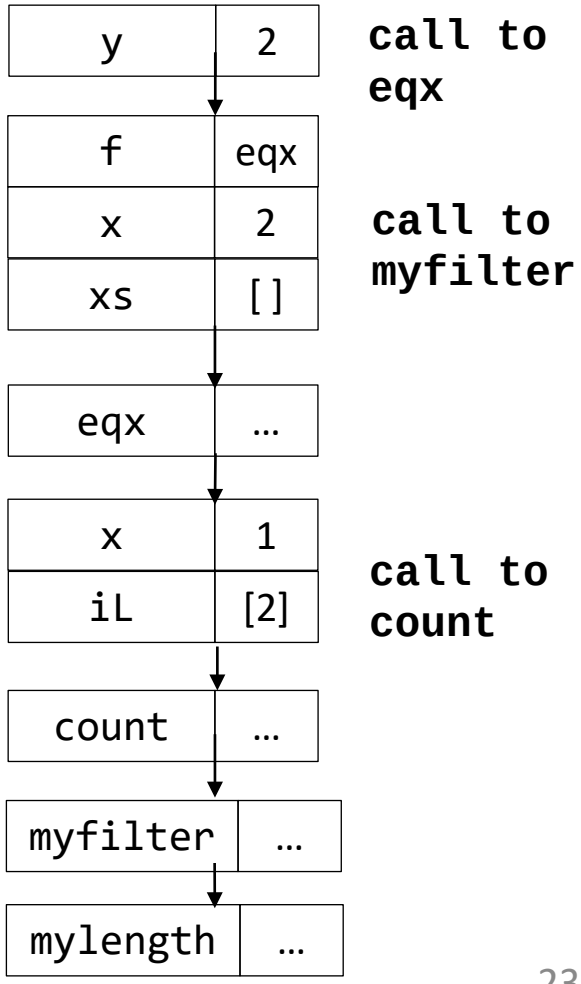
The output using static scope rules:

```
in main program -- n =
in chicken -- m =
in chicken -- n =
in egg -- n =
```

Static Scoping - Why it matters?

Example:

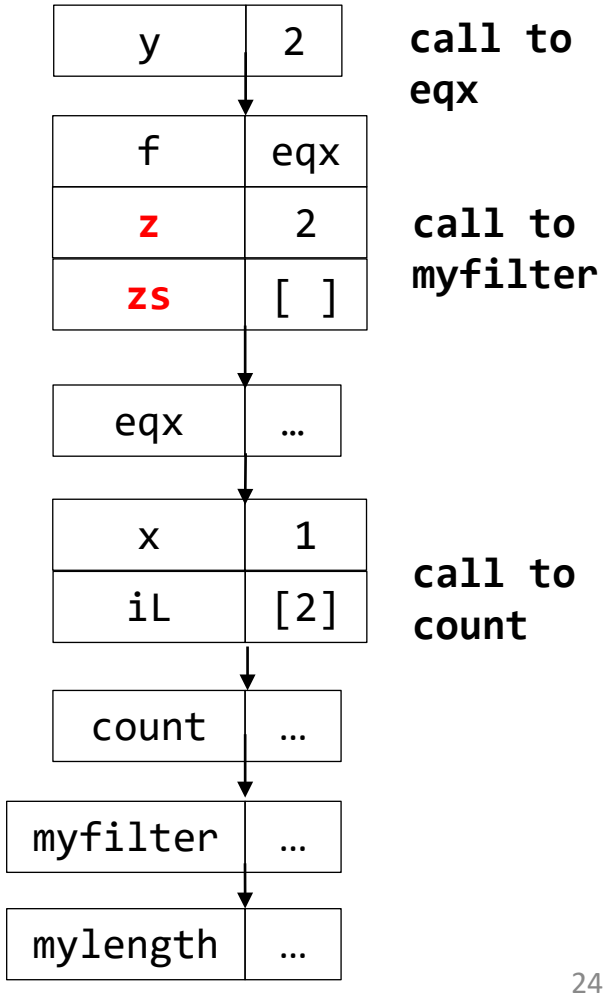
```
mylength [] = 0
mylength (x:xs)= 1 + mylength(xs)

myfilter f [] = []
myfilter f (x:xs) | (f x) = x:(myfilter f xs)
                  | otherwise = myfilter f xs

count x iL = let
               eqx y = (x==y)
             in
               mylength (myfilter eqx iL)

result = count 1 [2]
```



Static Scoping - Why it matters?

Example:

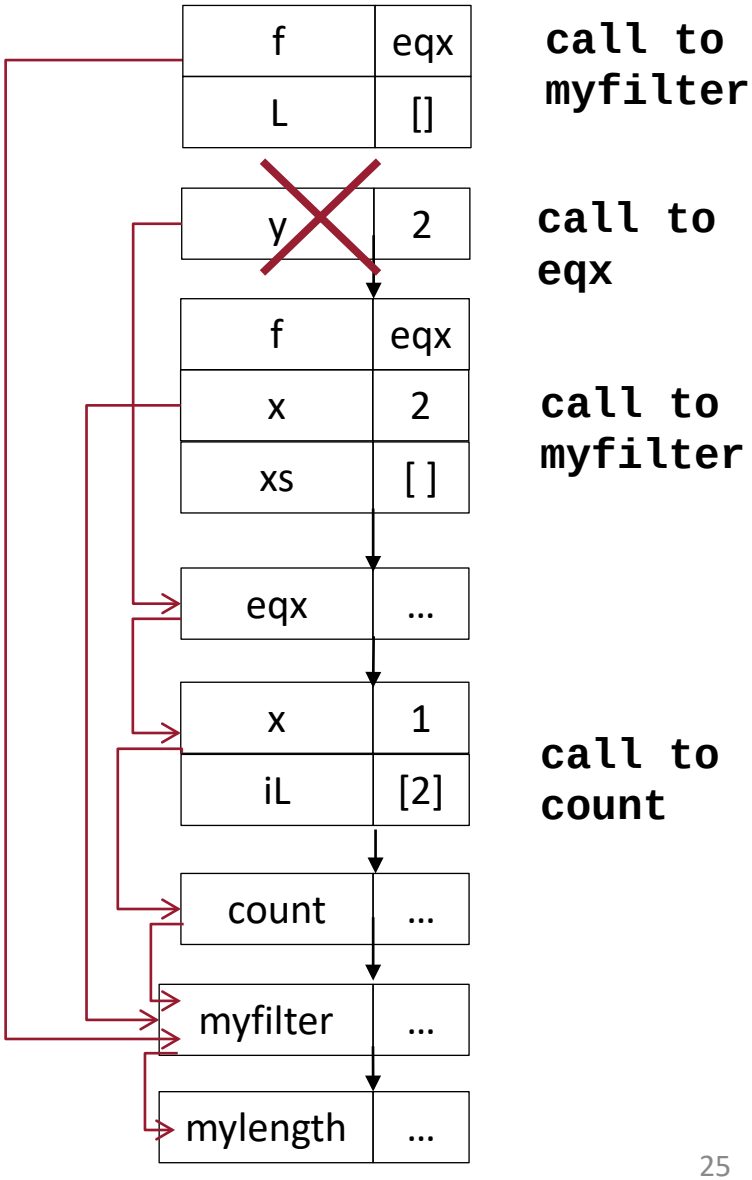
```
mylength [] = 0
mylength (x:xs)= 1 + mylength(xs)

myfilter f [] = []
myfilter f (z:zs) | (f z) = z:(myfilter f zs)
| otherwise = myfilter f zs

count x iL = let
               eqx y = (x==y)
             in
               mylength (myfilter eqx iL)

result = count 1 [2]
```



Static Scoping - Why it matters?

Example:

```
mylength [] = 0
mylength (x:xs)= 1 + mylength(xs)

myfilter f [] = []
myfilter f (x:xs) | (f x) = x:(myfilter f xs)
                  | otherwise = myfilter f xs

count x iL = let
               eqx y = (x==y)
             in
               mylength (myfilter eqx iL)

result = count 1 [2]
```



- Decades ago, both might have been considered reasonable, but now we know static scope makes much more sense
- Therefore, language designers have mostly concluded that the static scope rule is preferable to the dynamic scope rule.

Static Scoping - Why it matters?

- Function meaning does not depend on variable names used.
- Functions can be type-checked and reasoned about where they are defined

Example:

```
mylength [] = 0
mylength (x:xs)= 1 + mylength(xs)

myfilter f [] = []
myfilter f (x:xs) | (f x) = x:(myfilter f xs)
                  | otherwise = myfilter f xs

count x iL = let
               eqx y = (x==y)
             in
               mylength (myfilter eqx iL)

result = count 1 [2]
```

Higher Order Functions – Functions as First Class Values

- **First-class functions:** Can use them wherever we use values
  - Functions are values too
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, …

```
double x = 2*x
negate x = -1*x
f_tuple = (double, negate, double(negate 7))
```

- Most common use is as an argument / result of another function
  - Other function is called a higher-order function
  - Powerful way to factor out common functionality

Static Scope and Higher Order Functions

- The rule stays the same:
  - A function body is evaluated where the function body is defined; extended with the function argument
- Nothing changes to this rule when we take and return functions
  - Example:

```
f g =
      let
        x = 3
      in
        g 2
x = 4
h y = x + y
z = f h
```

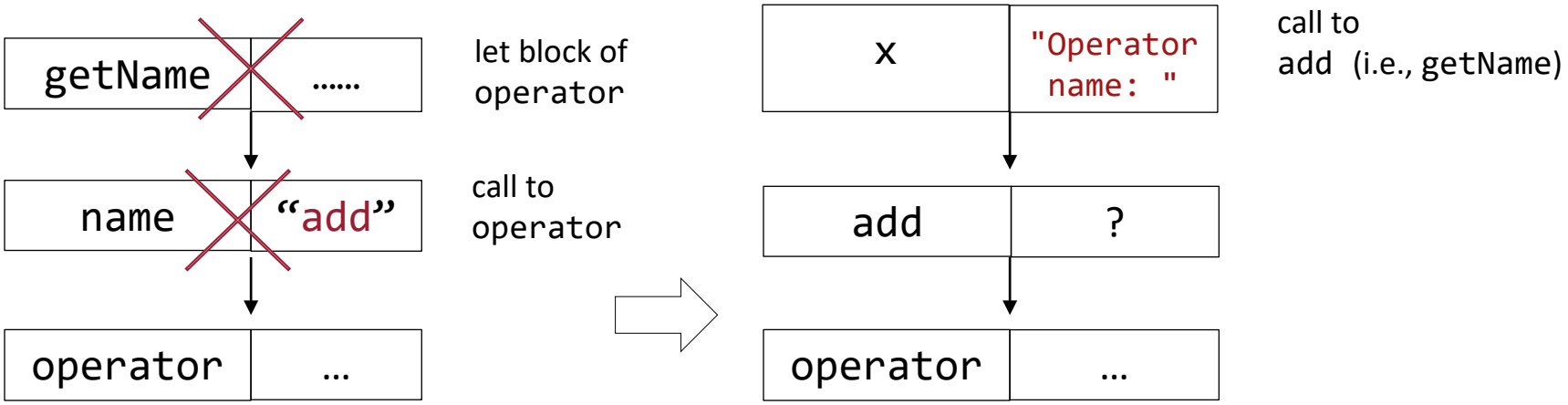
z evaluates to ?

Closures

Example:

```
operator name = let
                  getName x = x ++ name
                in
                  getName

add = operator "add"
result = add "Operator name: "
```

- How can functions be evaluated in old environments that aren't around anymore?
  - The language implementation keeps them around as necessary

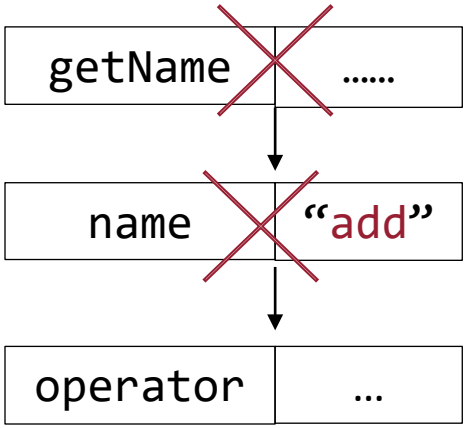


- The problem is: the activation record for "getName" is no longer on the stack and "name" is undefined.
- Therefore when a function is returned, it's **closure** is saved (in the heap) to remember it's referencing environment in effect when it was created.

Closures

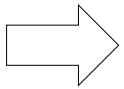
Example:

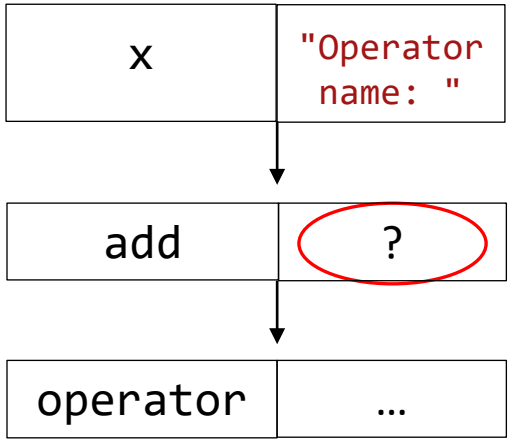
```
operator name = let
                  getName x = x ++ name
                in
                  getName

add = operator "add"
result = add "Operator name: "
```



let block of operator

call to operator





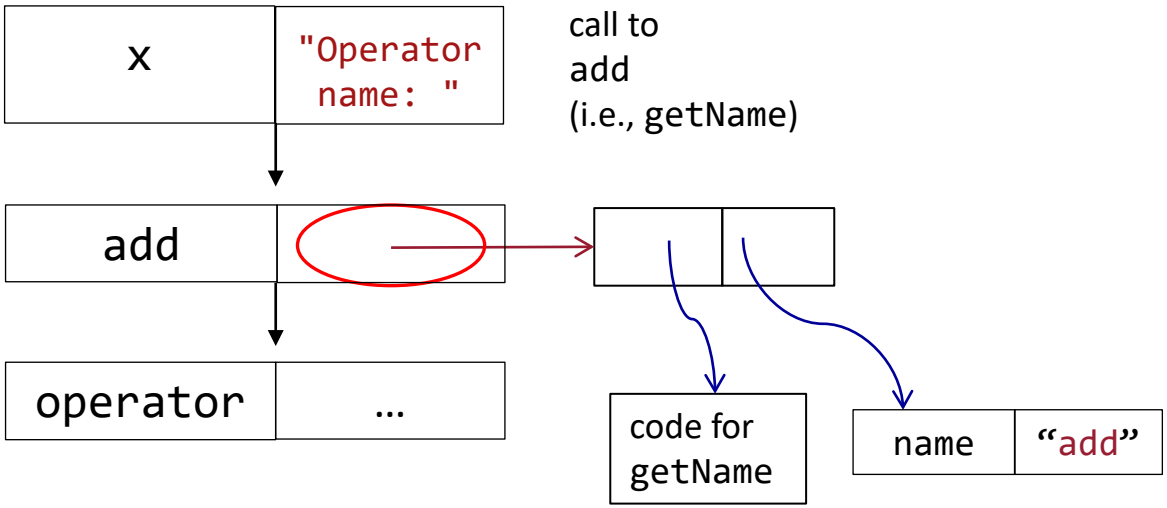
call to add (i.e., getName)

- The problem is: the activation record for "call to operator" is no longer on the stack ("name" is undefined) .
- Therefore when a function is returned, it's **closure** is saved (in the heap) to remember it's referencing environment in effect when it was created.

"add" is a variable with a closure value.

Closures

- Can define the semantics of functions as follows:
  - A function value has two parts
    - the code (obviously)
    - referencing environment that was current when the function was defined
  - This pair is called a **function closure**
  - A call evaluates the code part in the environment part (extended with the function argument)



Parameter Passing

- All our languages we covered so far had the same semantics for function calls:
  - Evaluate the function arguments.
  - Bind the argument names to the argument values (i.e., we create an AR in which the formal parameters are associated with the actual arguments passed in the call).
  - Evaluate the function body in the environment produced in step 2.

- This is called **pass by value** parameter passing, because the parameter names are bound to the values of the arguments.
  - Most common method in parameter passing

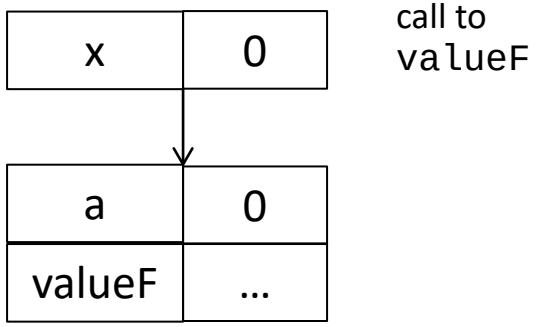
- There are several other ways to design parameter passing; some are used by popular languages today, but others are mostly of historical interest.

Parameter Passing

Pass by Value:

- Evaluate the actual argument expressions and find the values to the formal parameters.
- Example:

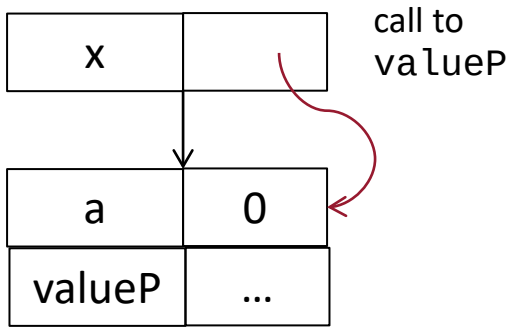
```
In C++ :
// Takes an int by value
void valueF(int x) { x = x + 1; }

int main() {
    int a = 0;

    valueF(a);
    cout << a << endl;  // Prints "0"
    return 0;
}
```



Parameter Passing

- How about the pointers passed to functions in C? Are they passed by value?

```
In C :
void valueP(int *x) { *x = *x + 1; }

int main() {
    int a = 0;

    valueP(&a);
    printf("%d",a);   // Prints "1"
    return 0;
}
```

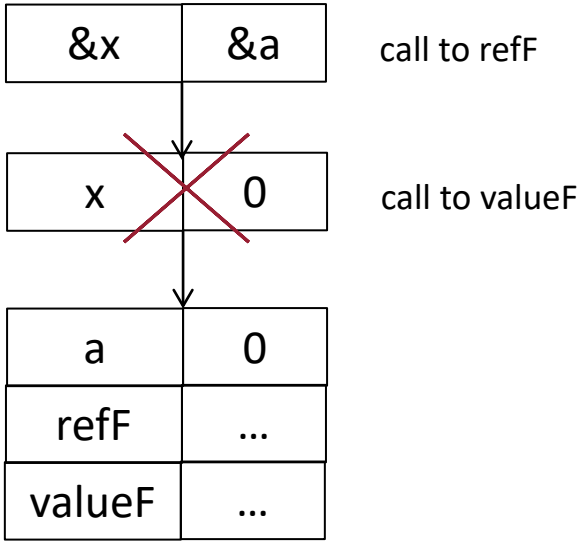


Parameter Passing

Pass by Reference:

- The formal arguments name is bound to the address of the passed value.
- Example:

```
In C++ :
// Takes an int by value
void valueF(int x) { x = x + 1; }

// Takes an int by reference.
void refF(int& x) { x = x + 1; }

int main() {
    int a = 0;

    valueF(a);
    cout << a << endl;  // Prints "0\n"

    refF(a);
    cout << a << endl;  // Prints "1\n"
    return 0;
}
```



Parameter Passing

How about in Python?

```
def f(x): x[0] =1
L=[0,1,2]
f(L)
```

vs.

```
def f(x): x =[1]
L=[0,1,2]
f(L)
```

In Python “object references are passed by value.”

Parameter Passing

Pass by Name:

- Code to evaluate the actual argument is bound to the formal parameter name.
- Examples:

```
function f (x,y) {
    x=x+1;
    return y[x]
}

f i j

will turn into :

{i=i+1;
return j[i]}
```

```
function f (x,y) {
    x=x+1;
    return y
}

f i  i-1

will turn into :

{i=i+1;
return (i-1) }
```

Not used in modern programming languages!

Suppose you have an actual argument, that is likely to be either very expensive to compute, or impossible to compute, and you only want to evaluate it if it is really needed.
Pass by name is ideal in such a case, because the argument expression is not evaluated, until it is really needed.

Parameter Passing

Other parameter passing modes:

- Pass by result.
- Pass by value result.

We won't talk about these.