Scope

Reusing Names

- Scope is trivial when all names are unique
- ```
  let square a = a * a;;
  let double b = b + b;;
  ```
- But in modern languages, names are often reused over and over:

```
let square n = n * n;;
let double n = n + n;;
```

- How can this work?
```

Outline

- Definitions and scope
- Scoping with blocks
- Scoping with labeled namespaces
- Scoping with primitive namespaces
- Dynamic scoping
- Separate compilation

Definitions

- When there are different variables with the same name, there are different possible bindings for that name
- Scope of variables, type names, constant names, function names, etc.
- A definition is anything that establishes a possible binding for a name

Examples

```
let square n = n * n;;
let square square = square * square;;
```

```
typedef struct {
      int i;
      float f;
} mystruct;

mystruct ms = {5, 3.0};

{     int i = ms.i;
      int ms = i;
}

cout << ms.i;
```

Examples

```
class myClass {
     public int i;
     public double f;
     public myClass(int i, double f) {
          this.i=i; this.f=f;
     }
}
public class scope {
     public static void Main( ) {
          myClass mc = new myClass(5,
3.0);
          int i = mc.i;
          int f = (int) mc.f;
          Console.WriteLine( mc.i );
     }
}
```

Scope

- There may be more than one definition for a given name
- Each occurrence of the name (other than a definition) has to be bound according to one of its definitions
- An occurrence of a name is *in the scope of* a given definition of that name whenever that definition governs the binding for that occurrence

Examples

```
let square square = square * square;

square 3;
Int = 9
```

- Each occurrence must be bound using one of the definitions
- Which one?
- There are many different ways to solve this scoping problem

Why Scope is Necessary

- Control access to specific bindings
- Information hiding - restrict access to only those bindings necessary
- Parnas Principles on Information Hiding
  - *Implementer* has only information necessary to complete module and nothing more. Has no knowledge of user module internals.
  - *User* has only information necessary to complete module and nothing more. Has no knowledge of implementation module internals.

Exercise 1 – C User & Implementer

a)

```
typedef struct {
   double s[10];
   int    top;
} Stack;

void initialize(Stack &stk){
   stk.top = -1;
}
void push(Stack &stk, double e){
   stk.s[++stk.top] = e;
}
double pop( Stack &stk) {
   return stk.s[stk.top--];
}
```

b)

```
void initialize();
void push(double e);
double pop();
void main() {
   Stack A;
   initialize(A);
   push(A, 3.14);
   push(A, -13.0);
   A.top = -12;
   cout << pop(A);
}
```

1. Indicate which of a) or b) define *user* and *implementer* parts.
2. *Parnas* principles are violated. Where and how?

Exercise 1 – C User & Implementer

```
typedef struct {
   double s[10];
   int    top;
} Stack;

Stack stk;

void initialize() {
   stk.top = -1;
}
void push(double e) {
   stk.s[++stk.top] = e;
}
double pop() {
  return stk.s[stk.top--];
}
```

```
void initialize();
void push(double e);
double pop();

void main() {
   initialize();
   push(3.14);
   push(-13.0);
   cout << pop();
}
```

- Is this a better information hiding solution than the previous? Why or why not?

Exercise 1 – C++ User & Implementer

```
class Stack  {
 private:
  double    s[10];
  int       top;
 public:
  Stack();
  double pop();
  void push( double e );
 };
 Stack::Stack() {
     top = -1;
 }
 double Stack::pop() {
    return s[top--];
 }
 void Stack::push(double e){
     s[ ++top ] = e;
 };
```

```
class Stack  {
 public:
  Stack();
  double pop();
  void push( double e );
};
void main() {
  Stack *s1 = new Stack();
  s1->push(3.14);
  s1->push(-13.0);
  cout << s1->pop();
}
```

1. What names are visible to the user above?
2. A better information hiding solution? Why or why not?

Exercise 1 – C# User & Implementer

```
class Stack  {
 private
   double []s =
            new double[10];
 private int top;

 public Stack() {
     top = -1;
 }

 public double pop() {
     return s[top--];
 }

 public
   void push(double e){
     s[ ++top ] = e;
   }
}
```

```
public class Ex1  {
 static void Main(){
   Stack s1 = new Stack();
   s1.push(3.14);
   s1.push(-13.0);

   Console.Write(s1.pop());
 }
}
```

1. What names are visible to the user above?
2. A better information hiding solution? Why or why not?

Outline

- Definitions and scope
- Scoping with blocks
- Scoping with labeled namespaces
- Scoping with primitive namespaces
- Dynamic scoping
- Separate compilation

Blocks

- A block is any language construct that contains definitions
- And also contains the region of the program where those definitions apply

F#

```
let x = 1
let y = 2
in
  x+y
```

C++ not C#

```
{     int x = 1;
      int y = 2;

      {     int y = x+3;

            cout << x+y;

      }
}
```

Different F# Blocks

- The **`let`** is just a scoping block:

```
let x = 1
let y = 2
in
   x+y;;
```

- A **`function`** definition includes a block:

```
let cube x = x*x*x;;
```

- Multiple alternatives have multiple blocks:

```
let f L = match L with
          | h::Nil -> h
          | h::t   -> h
```

C# Blocks

- In C# and other C-like languages, you can combine statements into one *compound statement* using **{** and **}**
- A compound statement can also serve as a scoping block:

```
int i;
for(i=0; i<3; i++)
{   int i = i+i;
    System.Console.WriteLine( i );
}
```

- What is the effect above in C#?

  Answer: Syntax error redefining variable i.

Nesting

- What happens if a block contains another block, and both have definitions of the same name?
- F# example: what is the value of n printed?
- C++: what is the value of n printed?

```
let n = 1
in
   let n = 2
   in
      n
```

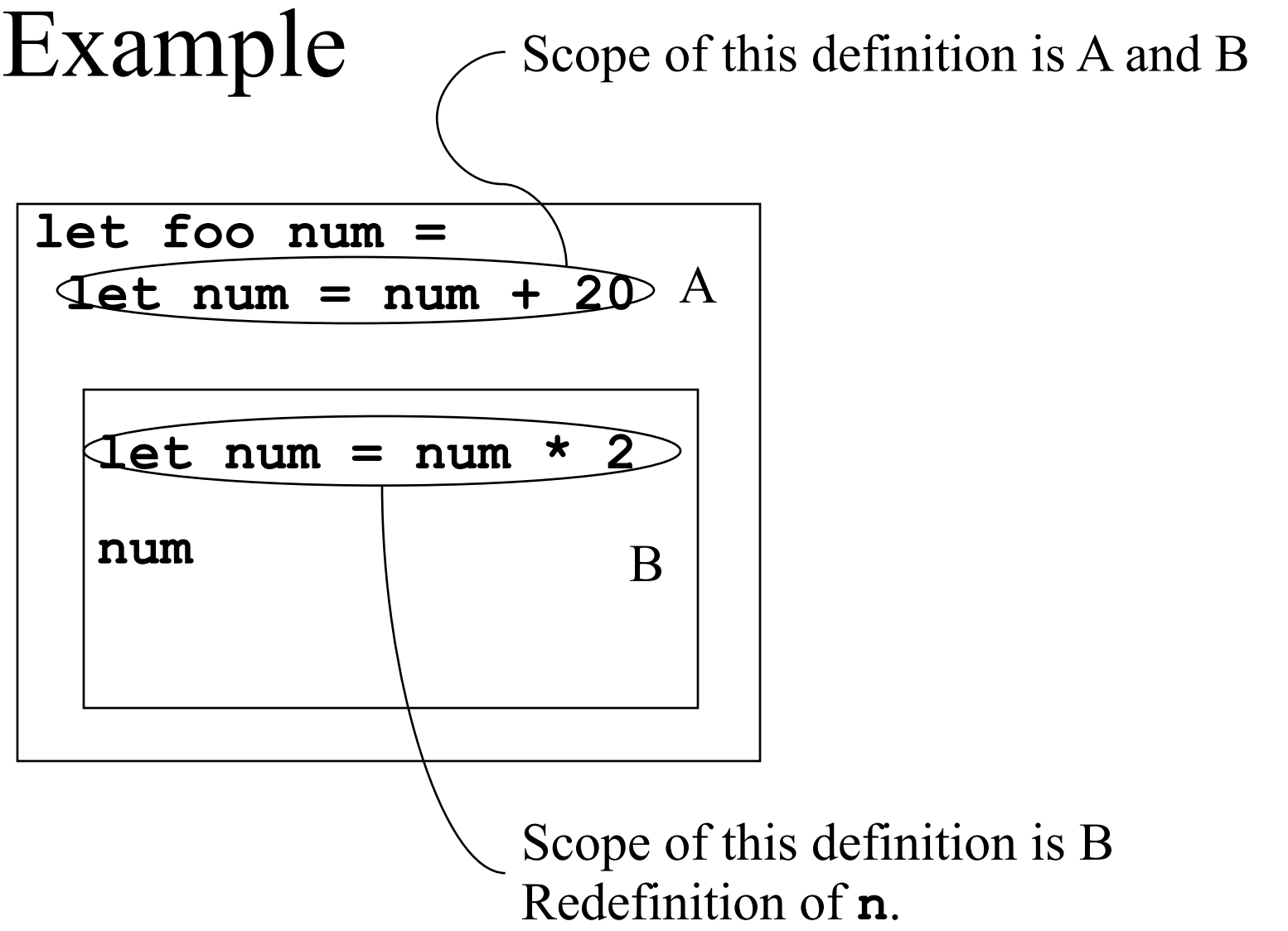
```
int n = 1;
{  int n = 2;
   cout << n;
}
```

Classic Block Scope Rule

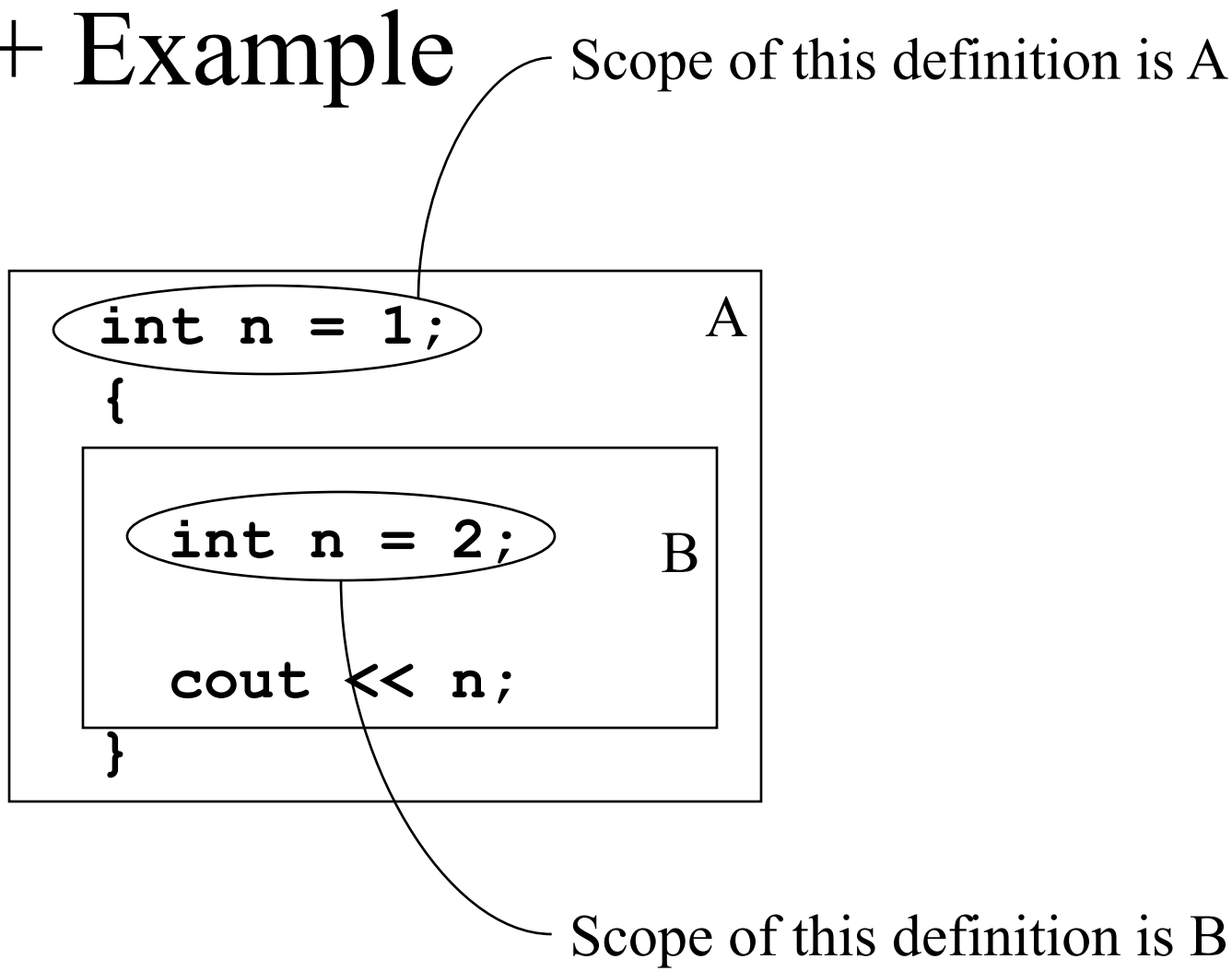
- The scope of a definition is the block containing that definition, from the point of definition to the end of the block, minus the scopes of any redefinitions of the same name in interior blocks

- That is F#'s rule; most statically scoped, block-structured languages use this or some minor variation

F# Example


Scope of this definition is A and B
let foo num =
let num = num + 20
A
let num = num * 2
num
B
Scope of this definition is B
Redefinition of n.

C++ Example


Scope of this definition is A and B
int n = 1;
A
{
int n = 2;
B
cout << n;
}
Scope of this definition is B

Outline

- Definitions and scope
- Scoping with blocks
- Scoping with labeled namespaces
- Scoping with primitive namespaces
- Dynamic scoping
- Separate compilation

Labeled Namespaces

- Namespaces that are just namespaces:
  - C++ , C# **`namespace`**
  - Modula-3 **`module`**
  - Ada **`package`**
  - Java **`package`**
- Namespaces that serve other purposes too:
  - Class definitions in class-based object-oriented languages

Example

```
public class Month {
  public static int min = 1;
  public static int max = 12;
  …
}
```

- The variables **`min`** and **`max`** would be visible within the rest of the class
- Also accessible from outside, as **`Month.min`** and **`Month.max`**
- Classes serve a different purpose too

Namespace Advantages

- Two conflicting goals:
  - Use memorable, simple names like **`max`**
  - For globally accessible things, use uncommon names like **`maxSupplierBid`**, names that will not conflict with other parts of the program

- With namespaces, you can accomplish both:
  - Within the namespace, you can use **`max`**
  - From outside, **`SupplierBid.max`**

Namespace Refinement

- Most namespace constructs have some way to allow part of the namespace to be kept private
- Often a good *information hiding* technique
- Programs are more maintainable when scopes are small
- For example, *abstract data types* reveal a strict interface while hiding implementation details…

Two Approaches

- Languages, such as C++, the namespace specifies the visibility of its components
- Other languages, such as ML, a separate construct defines the interface to a namespace (a *signature* in ML). The STACK *structure* has the *signature* at right.

```
structure STACK = struct
    fun pop (h::t) = h;
    fun remove (h::t)=t;
    fun push h t = h::t;
end;
```

```
signature STACK = sig
   val pop : 'a list -> 'a
   val push : 'a -> 'a list -> 'a list
   val remove : 'a list -> 'a list
end;
```

- And some languages, such as Ada and C#, combine the two approaches

Abstract Data Type

A **type** with associated operations exposed but implementation hidden.

namespace dictionary contains
private:
a constant definition for **initialSize**
a type definition for **hashTable**
a function definition for **hash**
a function definition for **reallocate**
public:
a function definition for **create**
a function definition for **insert**
a function definition for **search**
a function definition for **delete**
end namespace

Implementation definitions hidden

User interface definitions visible

Separate Interface

interface dictionary contains
a function type definition for **create**
a function type definition for **insert**
a function type definition for **search**
a function type definition for **delete**
end interface

namespace myDictionary implements dictionary contains
a constant definition for **initialSize**
a type definition for **hashTable**
a function definition for **hash**
a function definition for **reallocate**
a function definition for **create**
a function definition for **insert**
a function definition for **search**
a function definition for **delete**
end namespace

Example: An Abstract Data Type

```
class Stack {
        private LinkedList<Object> data;
```

```
public void push ( Object o ) {
   data.Add( o );
}
```

Implementation definitions hidden

User interface definitions visible

C# namespace example in same file.

```
namespace NameSpace1 {
    public class MyClass {
      static void Main()   {
         NameSpace2.NameSpace2Class.SayHello();
      }
    }
}

namespace NameSpace2 {
    public class NameSpace2Class {
        public static void SayHello()   {
            System.Console.WriteLine("Hello");
        }
    }
}
```

C# namespace example two files (Visual Studio)

```
using NameSpace2;
namespace NameSpace1 {
    public class MyClass {
        static void Main()   {
            NameSpace2.NameSpace2Class.SayHello();
        }
    }
}
```

```
namespace NameSpace2 {
    public class NameSpace2Class {
        public static void SayHello()   {
            System.Console.WriteLine("Hello");
        }
    }
}
```

NameSpace2 C# Class Library

NameSpace1 C# Console Application, add reference to *NameSpace2.dll*

Exercise 1.7 What corresponds in Java to *namespace* and *using?*

```
using NameSpace2;
namespace NameSpace1 {
    public class MyClass {
        static void Main()   {
            NameSpace2.NameSpace2Class.SayHello();
        }
    }
}
```

```
namespace NameSpace2 {
    public class NameSpace2Class {
        public static void SayHello()   {
            System.Console.WriteLine("Hello");
        }
    }
}
```

Outline

- Definitions and scope
- Scoping with blocks
- Scoping with labeled namespaces
- Scoping with primitive namespaces
- Dynamic scoping
- Separate compilation

Do Not Try This At Home

```
let int = 4;;
int : int = 4
```

- It is *legal* to have a variable named **`int`**
- F# is not confused but you might be.
- You can even do this:

```
let int int = int * int;;
  val int : int:int -> int
int 3;;
  val it : int = 9
```

Primitive Namespaces

- F#'s syntax keeps types and expressions separated
- F# always knows whether it is looking for a type or for something else
- There is a separate namespace for types

```
let int (int:int) = int * int;;
```

These are in the ordinary namespace

These are in the namespace for types

Primitive Namespaces

- Not explicitly created using the language (like primitive types)
- They are part of the language definition
- Some languages have several separate primitive namespaces
- C#: namespaces, types, methods, fields, and statement labels are in separate namespaces

Exercise 2 – Valid Java?

Reuse as package, class, attribute, method, parameter, and label name. Is it valid?

```
package Reuse;
class Reuse {
     Reuse Reuse;
     Reuse Reuse(Reuse Reuse) {
        Reuse:
         for(;;)
            if(Reuse.Reuse(Reuse) == Reuse)
               goto Reuse;
        return Reuse;
      }
}
```

Exercise 2 – Valid C#?

Reuse as namespace, class, parameter, and label name. Is it valid?

```
namespace Reuse {
  class Reuse {
     Reuse r;
     Reuse reuse(Reuse Reuse) {
       Reuse:
         for (; ; )
            if (Reuse.reuse(Reuse) == Reuse)
               goto Reuse;
            else return Reuse;

       }
    }
}
```

Outline

- Definitions and scope
- Scoping with blocks
- Scoping with labeled namespaces
- Scoping with primitive namespaces
- Dynamic scoping
- Separate compilation

When Is Scoping Resolved?

- All scoping tools seen so far are *static*
- Answer the question (whether a given occurrence of a name is in the scope of a given definition) at compile time
- Some languages postpone the decision until runtime: *dynamic scoping*

Dynamic Scoping

- Each function has an environment of definition
- If a name that occurs in a function is not found in its environment, its *caller's* environment is searched
- And if not found there, the search continues back through the chain of callers
- This generates a rather odd scope rule…

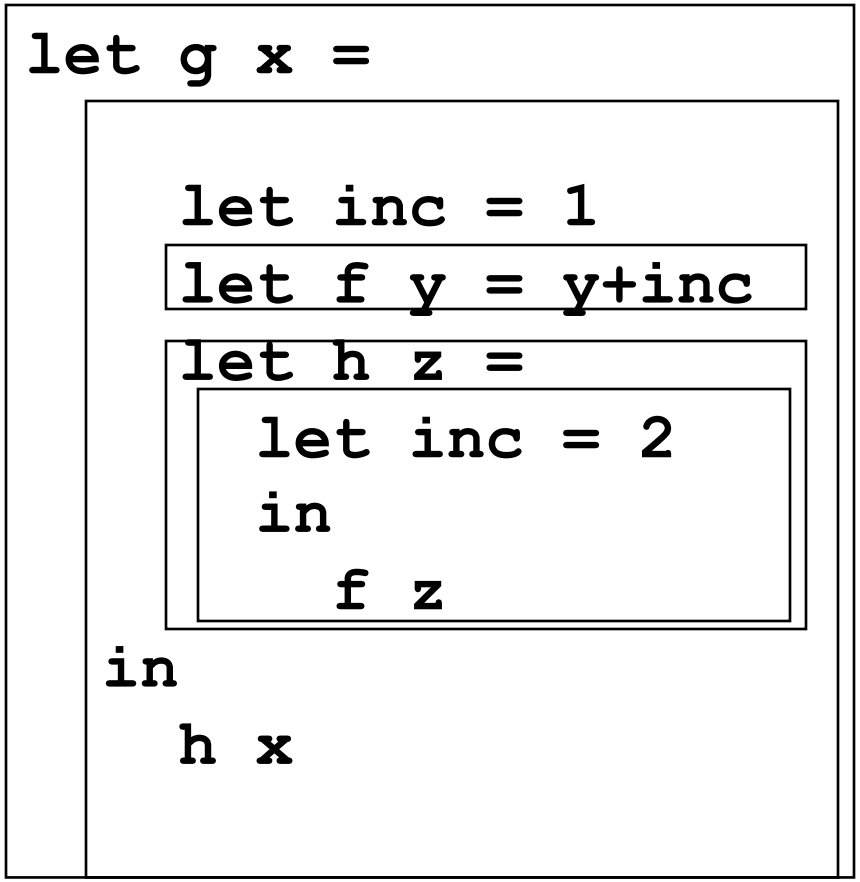
Classic Dynamic Scope Rule

- The scope of a definition is the function containing that definition, from the point of definition to the end of the function
- Along with any functions when they are called (even indirectly) from within that scope
- Minus the scopes of any redefinitions of the same name in those called functions

Static Versus Dynamic

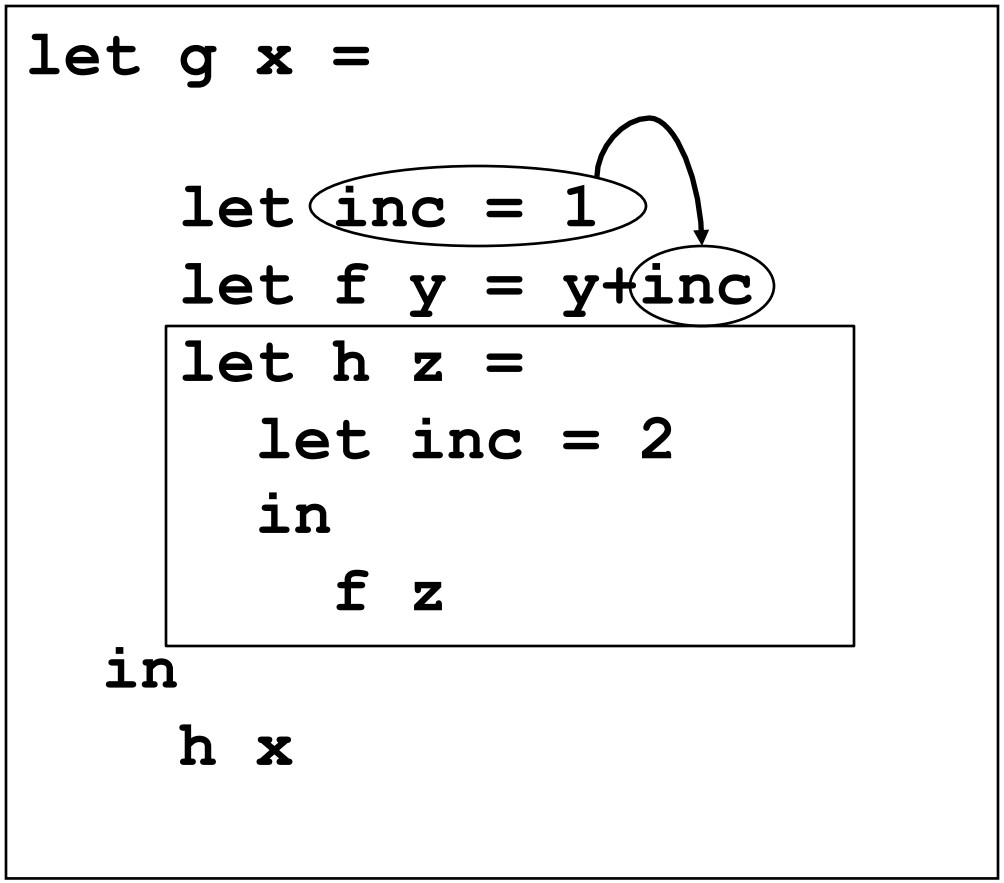
- The non-local environment is accessed by a link that points from one environment back to its enclosing environment.
- Both specify *scope holes*—places where a scope does not reach because of redefinitions
- The static rule specifies regions of program text, so environment can be determined at compile time.
- The environment **static** link points to the enclosing text.
- The dynamic rule specifies runtime events that determine the *enclosing* environment.
- Generally, the **dynamic** link points to the environment of the *calling* function.

Exercise 2.5



What is the value of `g(5)` using F#'s classic block static scope rule?

Block Scope (Static)



With block scope, the reference to **`inc`** is bound to the previous definition in the same block. **`f`**'s caller's environment, **`h`**, is inaccessible. The definition of **`inc`** in accessible to **`f`** is **`g`**'s.

`g 5` = 6 in F#

Dynamic Scope

Dynamic scope binds **`inc`** reference to the definition in the caller's environment. Effect at right is function **`f`** executes in caller's **`h`** environment. **`g(5)`** = 7 using dynamic scope

```
let g x =

    let inc = 1
    let f y = y+inc
    let h z =
      let inc = 2
      in
        f z
  in
    h x
```

```
let g x =
  let inc = 1;
    fun h z =
      let inc = 2;
       fun f y = y+inc;
      in
        f z
  in
    h x
```

C Static versus Dynamic Scope

```
int vue = 2;

int twice (int f(int), int vue) {
  return f (f (vue));
}

int f1( int x ) { return x * vue; }
```

twice (f1, 3); -> 12 statically scoped.

twice (f1, 3); -> 27 dynamically scoped.

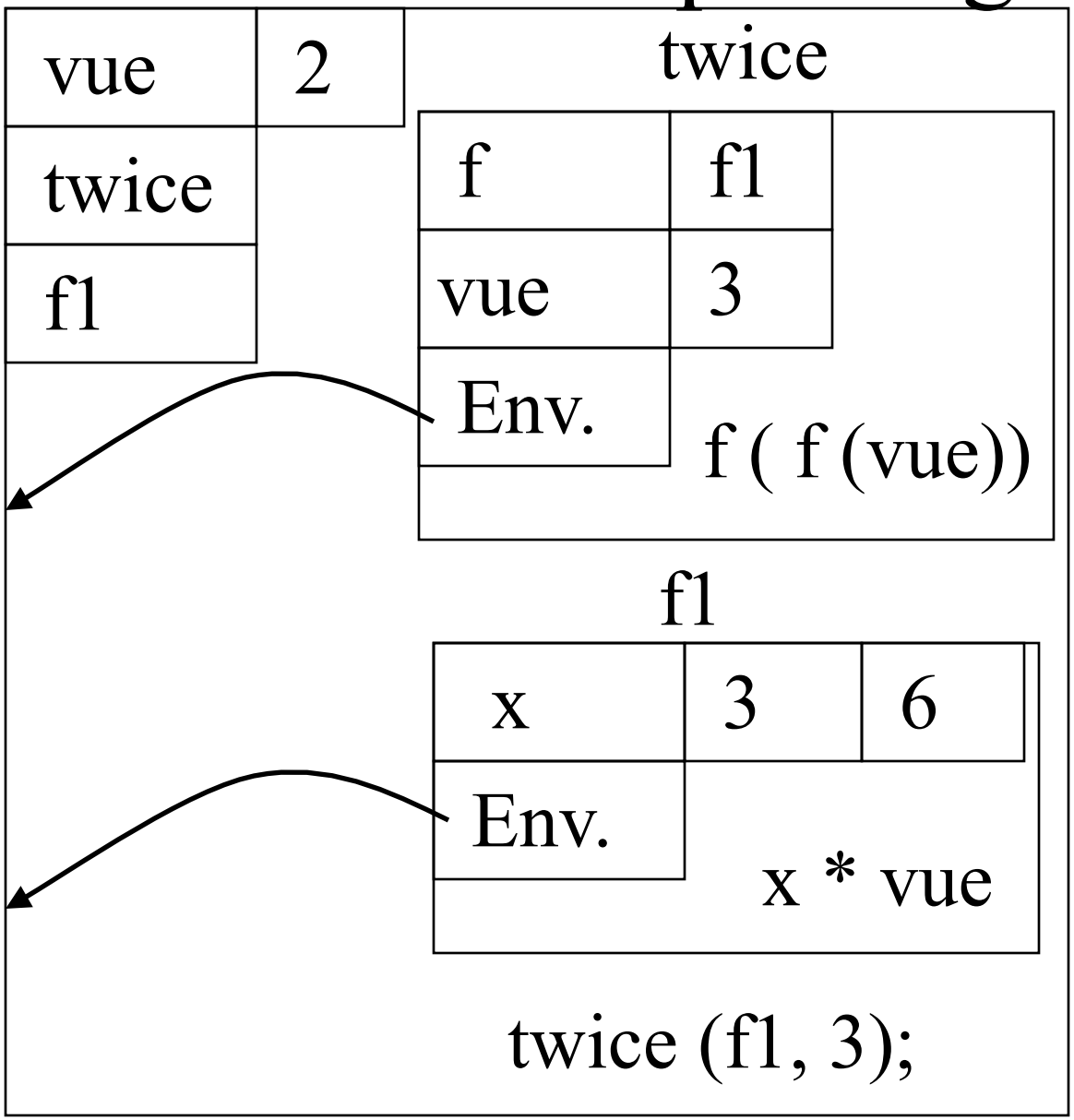
C Static versus Dynamic Scope

```
int vue = 2;
int twice( int f(int), int vue) {
  return f (f (vue)); }
int f1(int x) { return x * vue;}
twice (f1, 3 );                    Static
```

```
int vue = 2;
 int twice( int f(int), int vue) {
  return f (f (vue)); }
 int f1(int x) { return x * vue;}

twice (f1, 3);                     Dynamic
```

C Static Scope Diagram



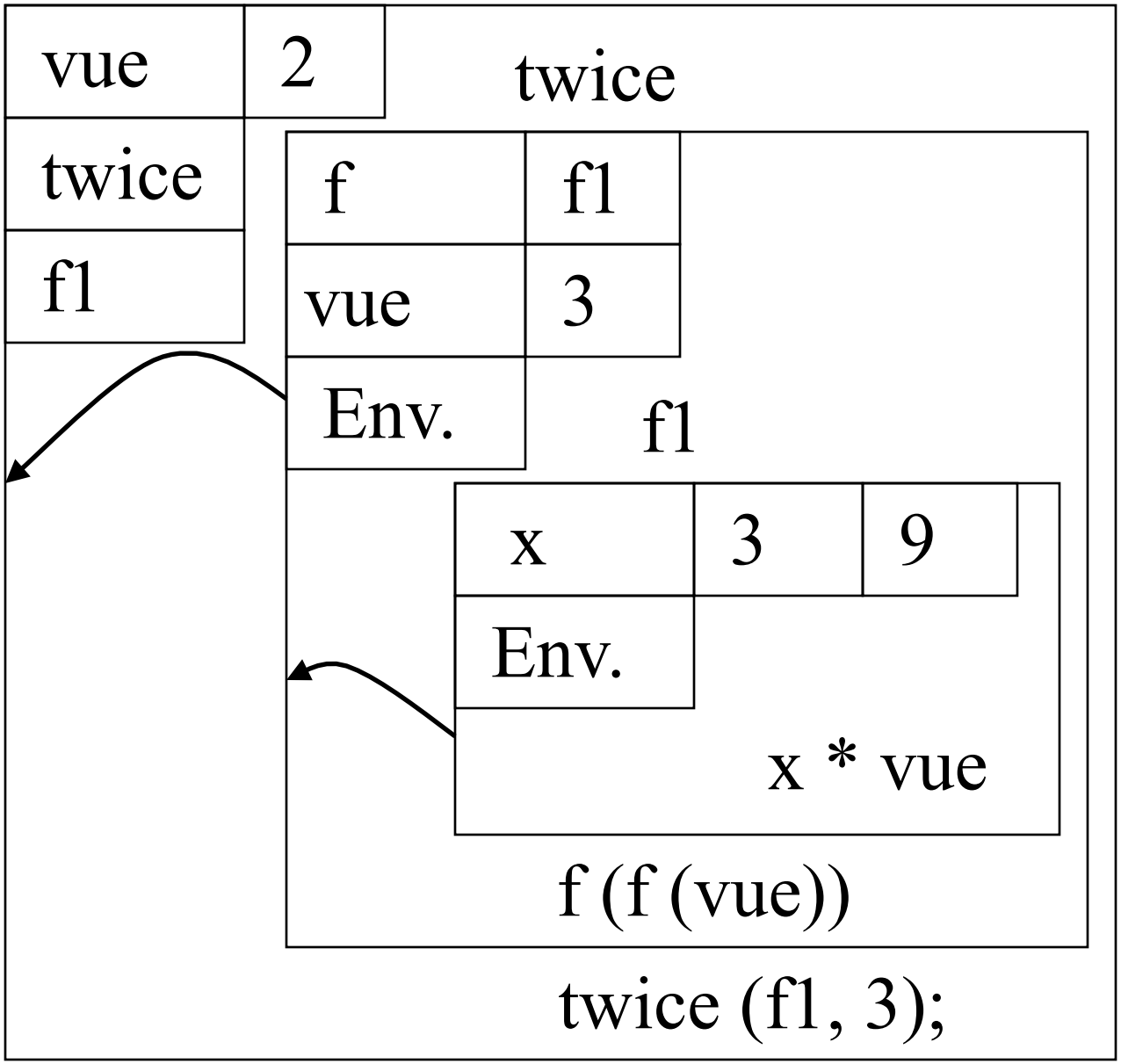

```
int vue = 2;

int twice(int f(int),
          int vue)
{ return f(f(vue));}

int f1(int x)
{ return x*vue; }
```

twice (f1, 3);
returns 12

C Dynamic Scope Diagram




```
int vue = 2;
                              3
int twice(int f(int),
              int vue)
{ return f(f(vue));}

int f1(int x)
{ return x*vue; }
```

twice (f1, 3);
returns 27

Where It Arises

- Only in a few languages: some dialects of Lisp, APL and our F#/valN interpreter
- Available as an option in Common Lisp
- Drawbacks:
  - Difficult to implement efficiently
  - Creates large and complicated scopes, since scopes extend into called functions
  - Choice of variable name in caller can affect behavior of called function

Outline

- Definitions and scope
- Scoping with blocks
- Scoping with labeled namespaces
- Scoping with primitive namespaces
- Dynamic scoping
- Separate compilation

Separate Compilation

- Common to the classical sequence of language system steps used by C++, Fortran
- Parts are compiled separately, then linked together
- Scope issues extend to the linker: it needs to connect references to definitions across separate compilations
- Many languages have special support for this

C# Namespace approach

```
using NameSpace2;
namespace NameSpace1 {
    public class MyClass {
        static void Main()   {
            NameSpace2.NameSpace2Class.SayHello();
        }
    }
}
```

```
namespace NameSpace2 {
    public class NameSpace2Class {
        public static void SayHello()   {
            System.Console.WriteLine("Hello");
        }
    }
}
```

NameSpace2 C# Class Library

NameSpace1 C# Console Application, add reference to *NameSpace2.dll*

C Approach, Compiler Side

- Two different kinds of definitions:
  - Full definition `int x = 3;` **full.cpp**
  - Name and type only: a *declaration* in C

```
#include <iostream.h>
extern int x;
void main(void) {
  cout << x;
}
```

declaration.cpp

- Several files use same integer variable **x**:
  - Only one will have the full definition,
    int x = 3;
  - All others have the declaration
    extern int x;

C Approach, Linker Side

- When the linker runs, it treats a *declaration* as a reference to a name defined in some other file
- It expects to see exactly one full definition of that name
- Note that the declaration does not say where to find the definition—it just requires the linker to find it somewhere
- Using Microsoft Visual C++ to compile and link files *full.cpp* and *declaration.cpp* to executable *full.exe:*

```
cl full.cpp declaration.cpp
```

Older Fortran Approach, Compiler Side

- Older Fortran dialects used **`COMMON`** blocks
- All separate compilations define variables in the normal way
- All separate compilations give the same **`COMMON`** declaration: **`COMMON A,B,C`**

Older Fortran Approach, Linker Side

- The linker allocates just one block of memory for the **`COMMON`** variables: those from one compilation start at the same address as those from other compilations
- The linker does not use the local names
- If there is a **`COMMON A,B,C`** in one compilation and a **`COMMON X,Y,Z`** in another, **`A`** will be identified with **`X`**, **`B`** with **`Y`**, and **`C`** with **`Z`**

Modern Fortran Approach

- A **`MODULE`** can define data in one separate compilation
- A **`USE`** statement can import those definitions into another compilation
- **`USE`** says what module to use, but does not say what the definitions are
- So unlike the C approach, the Fortran compiler must at least look at the result of that separate compilation

Trends in Separate Compilation

- In recent languages, separate compilation is less separate than it used to be
  - C# classes can depend on each other circularly, so the C# compiler must be able to compile separate classes ***simultaneously***
  - ML is not really suitable for separate compilation at all, though CM (a separate tool in the SML system, the Compilation Manager) can do it for most ML programs

Conclusion

- Today: four approaches for scoping
  - Scoping with blocks
  - Scoping with labeled namespaces
  - Scoping with primitive namespaces
  - Dynamic scoping
- There are many variations, and most languages employ several at once
- Remember: names do not have scopes, definitions do!