

# Scope

# Reusing Names

- Scope is trivial if you have a unique name for everything:

```
fun square a = a * a;  
fun double b = b + b;
```

- But in modern languages, we often use the same name over and over:

```
fun square n = n * n;  
fun 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
- Not just variables: type names, constant names, function names, etc.
- A definition is anything that establishes a possible binding for a name

# Examples

```
fun square n = n * n;  
fun square square = square * square;  
  
const  
  Low = 1;  
  High = 10;  
type  
  Ints = array [Low..High] of Integer;  
var  
  X: Ints;
```

# 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

```
- fun square square = square * square;  
val square = fn : int -> int  
- square 3;  
val it = 9 : int
```

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

# 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

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

# Different ML Blocks

- The **let** is just a block: no other purpose
- A **fun** definition includes a block:

```
fun cube x = x*x*x;
```

- Multiple alternatives have multiple blocks:

```
fun f (a::b::_) = a+b  
|    f [a] = a  
|    f [] = 0;
```

- Each rule in a match is a block:

```
case x of (a, 0) => a | (_, b) => b
```

# Java Blocks

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

```
while (i < 0) {  
    int c = i*i*i;  
    p += c;  
    q += c;  
    i -= step;  
}
```

# Nesting

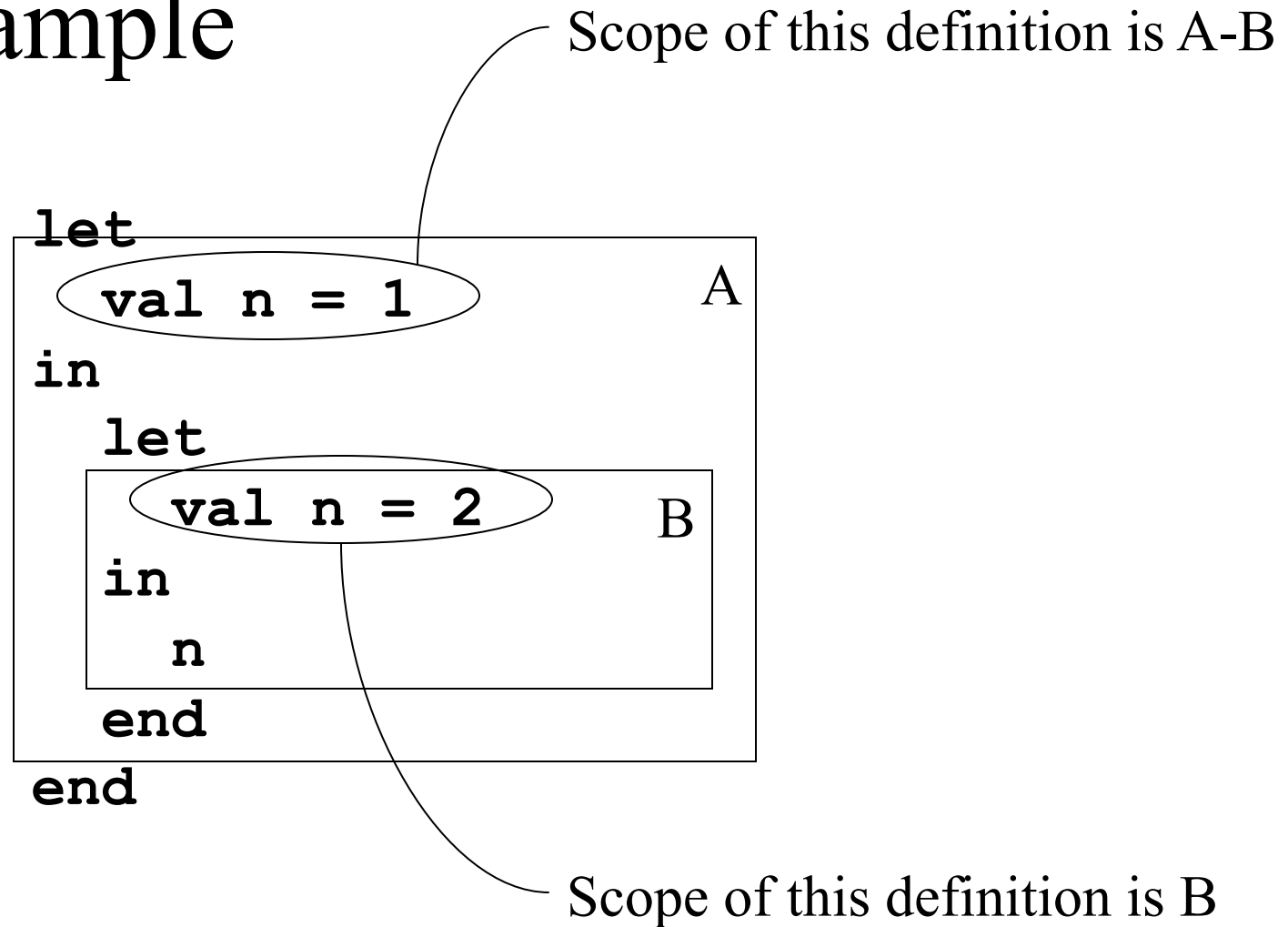
- What happens if a block contains another block, and both have definitions of the same name?
- ML example: what is the value of this expression:

```
let
  val n = 1
in
  let
    val n = 2
  in
    n
  end
end
```

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

# Example



# Outline

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

# Labeled Namespaces

- A labeled namespace is any language construct that contains definitions and a region of the program where those definitions apply, and also has a name that can be used to access those definitions from outside the construct
- ML has one called a *structure*...



# ML Structures

```
structure Fred = struct
  val a = 1;
  fun f x = x + a;
end;
```

- A little like a block: **a** can be used anywhere from definition to the end
- But the definitions are also available outside, using the structure name: **Fred.a** and **Fred.f**

# Other Labeled Namespaces

- Namespaces that are just namespaces:
  - 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...

# Example: An Abstract Data Type

**namespace 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**

Implementation  
definitions  
should be hidden

Interface definitions should be visible

# Two Approaches

- In some languages, like C++, the namespace specifies the visibility of its components
- In other languages, like ML, a separate construct defines the interface to a namespace (a *signature* in ML)
- And some languages, like Ada and Java, combine the two approaches

# Namespace Specifies Visibility

**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**



# 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
```

# 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

```
- val int = 3;  
val int = 3 : int
```

- It is legal to have a variable named **int**
- ML is not confused
- You can even do this (ML understands that **int\*int** is not a type here):

```
- fun f int = int*int;  
val f = fn : int -> int  
- f 3;  
val it = 9 : int
```

# Primitive Namespaces

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

```
fun f (int: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
- Java: packages, types, methods, variables, and statement labels are in separate namespaces

# 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 we have seen so far are static
- They 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 definitions
- 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 Vs. Dynamic

- The scope rules are similar
- Both talk about *scope holes*—places where a scope does not reach because of redefinitions
- But the static rule talks only about regions of program text, so it can be applied at compile time
- The dynamic rule talks about runtime events: “functions when they are called...”

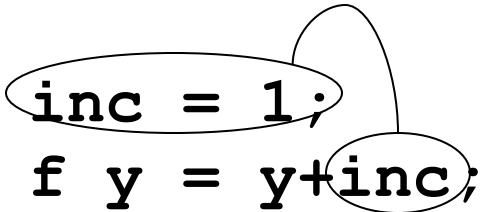
# Example

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

What is the value of  
**g 5** using ML's classic  
block scope rule?

# Block Scope (Static)

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

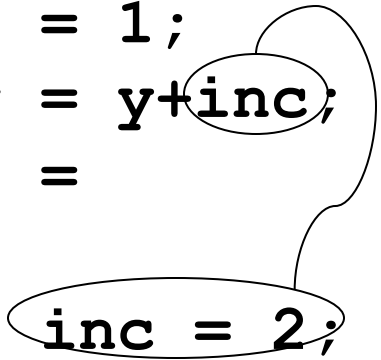


With block scope, the reference to **inc** is bound to the previous definition in the same block. The definition in **f**'s caller's environment is inaccessible.

**g 5 = 6** in ML

# Dynamic Scope

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



With dynamic scope,  
the reference to **inc** is  
bound to the definition  
in the caller's  
environment.

**g 5** = 7 if ML used  
dynamic scope

# Where It Arises

- Only in a few languages: some dialects of Lisp and APL
- 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

- We saw this in the classical sequence of language system steps
- 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 Approach, Compiler Side

- Two different kinds of definitions:
  - Full definition
  - Name and type only: a *declaration* in C-talk
- If several separate compilations want to use the 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

# 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
  - Java classes can depend on each other circularly, so the Java 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
- There are many variations, and most languages employ several at once
- Remember: names do not have scopes, definitions do!