Scope
Reusing Names

- Scope is trivial when all names are unique
  
  ```haskell
  fun square a = a * a;
  fun double b = b + b;
  ```

- But in modern languages, names are often reused over and over:
  
  ```haskell
  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
• Scope of 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;

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

mystruct ms = {5, 3.0};

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

cout << ms.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

- `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
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 = 1;
    cout << pop(A);
}

1. Indicate which of a) or b) define user and implementer parts.

2. Parnas principles are violated. Where and how?
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;
};

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 – ML User & Implementer

```ml
fun push x [] = [x]
|   push x L = x::L;
fun pop (h::t) = t;
fun top (h::t) = h;
```

ML discourages assignment (mutation) on existing bindings, so that:

```ml
val s = 5;
    s = 6;
result is false! This makes maintaining and altering the state of the
stack, through push and pop, difficult. Better solutions will be
discussed in Chapter 11.
```

- What is the result of each line below:

1. `val s = push 3 [];
2. `val s = push ~13 s;
3. `val s = pop s;
4. `top s;`
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

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

```plaintext
{ int x = 1;
  int y = 2;
  {
    cout << x+y;
  }
}
```
Different ML Blocks

- **The `let`** is just a scoping block: no other purpose

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

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

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

- **Multiple alternatives have multiple blocks:**

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

- **Each rule in a match is a block:**

  ```ml
  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 scoping block:

```java
int i;
for(i=0; i<3; i++)
{
    int i = i+i;
    System.out.println( i );
}
```

• What is the effect above?
Nesting

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

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

```cpp
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 ML’s rule; most statically scoped, block-structured languages use this or some minor variation.
Example

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

Scope of this definition is A and B

Scope of this definition is B
Example

```
Example

Scope of this definition is A and B

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

Chapter Ten
Modern Programming Languages
20
```
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

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

- A little like a block: `a` can be used anywhere inside definition to the **end**

- But the definitions are also available outside by:

  ```ml
  Fred.a;
  val it = 1 : int
  Fred.f 3;
  val it = 4 : int
  ```
ML Structures – STACK Example

pop, remove, push only accessible via STACK structure

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

val s = STACK.push 3 [] ; is [3]
val s = STACK.push 4 s ; is [4,3]
val s = STACK.remove s ; is [3]
STACK.pop s ; is 3
```
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

```java
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.

```plaintext
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 Java, combine the two approaches.
Example: An Abstract Data Type

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 should be hidden
Interface definitions should be 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
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 but you might be.
• 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

```ml
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, fields, and statement labels are in separate namespaces
Exercise 2 – Valid Java?

class Reuse {
    Reuse Reuse;
    Reuse Reuse(Reuse Reuse) {
        Reuse:
        for(;;)
            if(Reuse.Reuse(Reuse) == Reuse)
                break Reuse;
        return Reuse;
    }
}

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

```ml
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 static scope rule?
fun g x = 
 let 
   val inc = 1;
   fun f y = y+inc;
 in 
 h x 
end;

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. f’s caller’s environment, h, is inaccessible. The definition of inc in accessible to f is g’s.

g 5 = 6 in ML
Dynamic Scope

Dynamic scope binds \texttt{inc} reference to the definition in the caller’s environment. Effect at right is function \texttt{f} executes in caller’s \texttt{h} environment. \( g\ 5 = 7 \) using dynamic scope

\begin{verbatim}
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;
\end{verbatim}

\begin{verbatim}
fun g x = 
  let
    val inc = 1;
    fun h z = 
      let
        val inc = 2;
        fun f y = y+inc;
      in
        f z
      end;
  in
    h x
  end;
\end{verbatim}
ML Static versus Dynamic Scope

val vue = 2;
fun twice f vue = f (f vue);
fun f1 x = x * vue;

\[
\text{twice f1 3; } \rightarrow 12 \text{ statically scoped.}
\]
\[
\text{twice f1 3; } \rightarrow 27 \text{ dynamically scoped.}
\]
ML Static versus Dynamic Scope

val vue = 2;

fun twice f vue = f (f vue);

fun f1 x = x * vue;

twice f1 3;  Static

val vue = 2;

fun twice f vue = f (f vue);

fun f1 x = x * vue;

twice f1 3;  Dynamic
val vue = 2;
fun twice f vue = f (f vue);
fun f1 x = x*vue;
twice f1 3;
returns 12
ML Dynamic Scope Diagram

val vue = 2;
fun twice f vue = f (f vue);
fun f1 x = x * vue;
twice f1 3;
returns 27
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

```c
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
```

```c
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
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
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 27
```
Exercise 3

1. Give the static scoping diagram.
2. What is the result with static scoping?
3. Give the dynamic scoping diagram.
4. What is the result with dynamic scope?

```c
int vue = 4;

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

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

void main() { cout << twice( f1, 5); }
```
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

- 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 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;
}
```

- 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
  - 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
  - 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!