Lecture 6: Scopes and Symbol Tables

COSC 4316

Sam Houston State University

Partially based on *Introduction to Compiler Design* by Torben Aegidius Mogensen, University of Copenhagen

- "The scope of thrift is limitless" – Thomas Edison

Introduction

- An important concept in programming languages is the ability to *name* objects such as variables, functions and types.
- Each such named object will have a *declaration*, where the name is defined as a synonym for the object.
- This is called *binding*.
- Each name will also have a number of *uses*, where the name is used as a reference to the object to which it is bound.

Introduction

- The declaration of a name has a *scope*: a portion of the program where the name will be visible.
- The same name might be declared in several nested scopes.
- The scope of a declaration will be a sub-tree of the syntax tree and nested declarations will give rise to scopes that are nested subtrees.

Example

```
{
   int x = 1; int y = 2;
   {
      double x = 3.14159265;
      y += (int)x;
   }
   y += x;
}
```

- The first two statements declare integer variables **x** and **y** with scope until the closing brace in the last line.
- The second opening brace starts a new scope and declares a floating-point variable **x**.
- The original **x** variable is not visible until the inner scope ends.
- The assignment **y += (int)x;** adds 3 to **y**, so its new value is 5.
- The next assignment **y += x;** is in the outer scope, so the original **x** is used. The assignment will add 1 to **y**, which will have the final value 6.

Static Binding

- Scoping based on the structure of the syntax tree, as in the example, is called *static* or *lexical* binding.
- This is the most common scoping rule in modern programming languages.
- A few languages have *dynamic* binding, where the declaration that was most recently encountered during execution of the program defines the current use of the name.
- By its nature, dynamic binding can not be resolved at compile-time, so the techniques described here will have to be used at run-time if the language uses dynamic binding.
- In this course we will assume that static binding is used. (*You're welcome!*)

Symbol Tables

- The compiler needs to keep track of names and the objects these are bound to, so that any use of a name will be attributed correctly to its declaration.
- This is typically done using a *symbol table*.
- A symbol table is a table that binds names to information.
  - Consulted at almost every point during the compile process
  - Needs to be structured for speedy access during compile
  - Needs to reflect the organization of the source program
- We need a number of operations on symbol tables to accomplish this

Symbol Table Operations

- We need an *empty* symbol table, in which no name is defined.
- We need to be able to *insert* (or *bind*) a name to a piece of information. In case the name is already defined in the symbol table, the new binding takes precedence over the old.
- We need to be able to *look up* a name in a symbol table to find the information the name is bound to. If the name is not defined in the symbol table, we need to be told that.
- We need to be able to *enter* a new scope.
- We need to be able to *exit* a scope, reestablishing the symbol table to what it was before the scope was entered.

Symbol Table Implementation

- There are many ways to implement symbol tables, but the most important distinction between these is how scopes are handled.
  - This may be done using a *persistent* (or *functional*) data structure,
  - or it may be done using an *imperative* (or destructively-updated) data structure.
- In a *persistent* data structure, no operation on the structure will destroy it.
- Conceptually, a new copy is made of the data structure whenever an operation updates it, preserving the old structure unchanged.
- Thus it is trivial to reestablish the old symbol table when exiting a scope.
- In practice, only a small portion of the data structure is copied when a symbol table is updated, most is shared with the previous version.

Symbol Table Implementation

- In the ***imperative*** approach, only one copy of the symbol table exists, so explicit actions are required to store the information needed to restore the symbol table to a previous state.
- This can be done by using an auxiliary stack.
- When an update is made, the old binding of a name that is overwritten is recorded (pushed) on the auxiliary stack.
- When a new scope is entered, a marker is pushed on the auxiliary stack.
- When the scope is exited, the bindings on the auxiliary stack (down to the marker) are used to reestablish the old symbol table.
- Imperative symbol tables are natural to use if the compiler is written in an imperative language.

A Simple Imperative Symbol Table

- A simple imperative symbol table can be implemented as a stack with these operations:
  - ***empty***: An empty symbol table is an empty stack.
  - ***insert***: A new binding (name/information pair) is pushed on top of the stack.
  - ***lookup***: The stack is searched top-to-bottom until a matching name is found. The information paired with the name is then returned. If the bottom of the stack is reached, we instead return an error-indication.
  - ***enter***: We push a marker on the top of the stack.
  - ***exit***: We pop bindings from the stack until a marker is found. This is also popped from the stack.

A Simple Imperative Symbol Table

- Note that since the symbol table is itself a stack, we don't need the auxiliary stack mentioned previously.
- This is not a persistent data structure, since leaving a scope will destroy its symbol table.
- For simple languages, this won't matter, because a scope isn't needed again after it is exited.
- However, language features such as classes can require symbol tables to persist after their scope is exited.
- In these cases, a persistent symbol table must be used, or the needed parts of the symbol table must be copied and stored for later retrieval before exiting a scope.

Efficiency issues

- In the above, ***lookup*** is done by linear search, so the worst-case time for lookup is proportional to the size of the symbol table.
- This is mostly a problem in relation to libraries: It is quite common for a program to use libraries that define literally hundreds of names.
- A typical solution is ***hashing***: Names are hashed (processed) into integers, which are used to index an array.
- Given a large enough hash table, these lists will typically be very short, so the lookup time is essentially constant.

Efficiency issues

- Using hash tables complicates entering and exiting scopes.
- A typical way to handle this is for imperative implementations to use a single auxiliary stack (as indicated previously) to record all updates to the table so they can be undone in time proportional to the number of updates that were done in the local scope.

Shared or separate name spaces

- In some languages (like Pascal) a variable and a function in the same scope may have the same name, since the context of use will make it clear whether a variable or a function is used.
- We say that functions and variables have ***separate name spaces***, which means that defining a name in one space doesn't affect the same name in the other space.
- In other languages (such as C) the context can not (easily) distinguish variables from functions.
- Hence, declaring a local variable might hide a function declared in an outer scope or vice versa. These languages have a ***shared name space*** for variables and functions.

Shared or separate name spaces

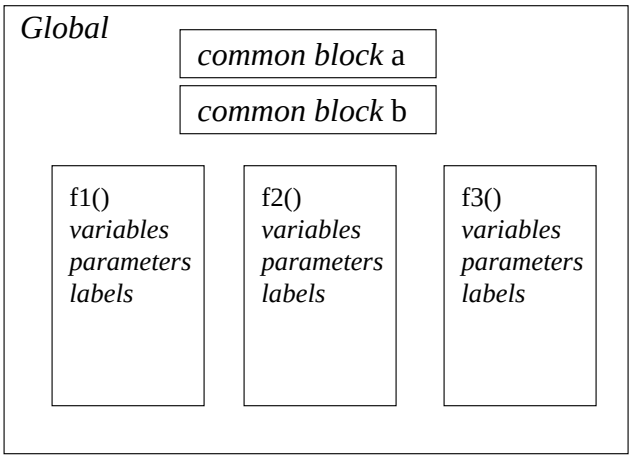
- Name spaces may be shared or separate for all the kinds of names that can appear in a program, *e.g.*, variables, functions, types, exceptions, constructors, classes, field selectors *etc*.
- Which name spaces are shared is language-dependent.
- Separate name spaces are easily implemented using one symbol table per name space, whereas shared name spaces naturally share a single symbol table.
- However, it is sometimes convenient to use a single symbol table even if there are separate name spaces.

Single symbol table with separate name spaces

- This can be done fairly easily by adding name-space indicators to the names.
- A name-space indicator can be a textual prefix to the name or it may be a tag that is paired with the name.
- In either case, a lookup in the symbol table must match both the name and the name-space indicator of the symbol that is looked up with the name and the name-space indicator of the entry in the table.

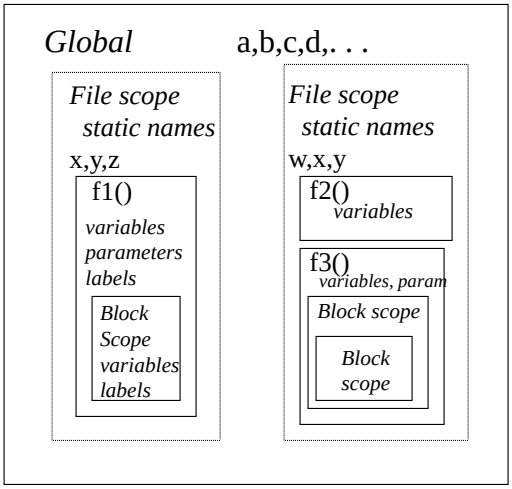
SOME EXAMPLE NAMESPACES

Fortran 77 Name Space



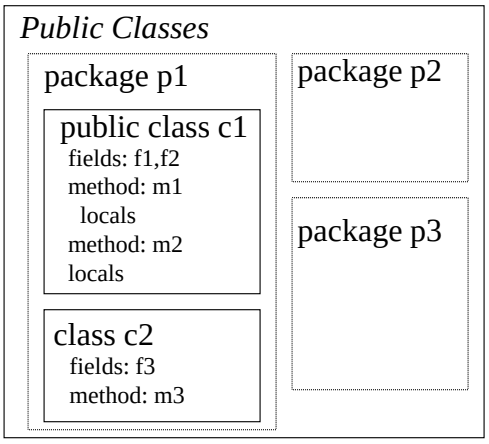
Global scope holds procedure names and common block names. Procedures have local variables parameters, labels and can import common blocks

C Name Space



- Global scope holds variables and functions
- No function nesting
- Block level scope introduces variables and labels
- File level scope with static variables that are not visible outside the file (global otherwise)

Java Name Space



- Limited global name space with only public classes
- Fields and methods in a public class can be public ➔ visible to classes in other packages
- Fields and methods in a class are visible to all classes in the same package unless declared private
- Class variables visible to all objects of the same class.

SYMBOL TABLES EXAMPLES

Scope and Parsing

```
func_decl  : FUNCTION NAME                  {EnterScope($2);}
               parameter decls stmts ;      {ExitScope($2); }

decl       :    name ':'  type              {Insert($1,$3); }

...
statements:   id := expression              {lookup($1);}
...
expression:       …
             id                             {lookup($1);}
```

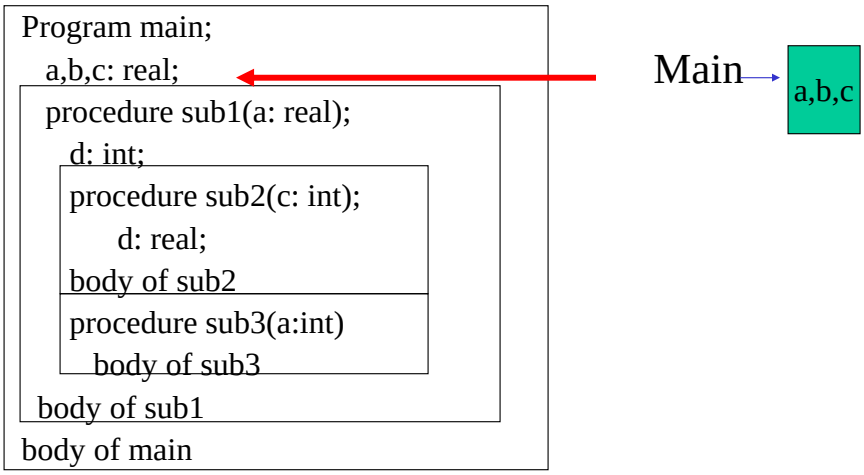
Note: This is a greatly simplified grammar including only the symbol table relevant productions.

Implementing the table

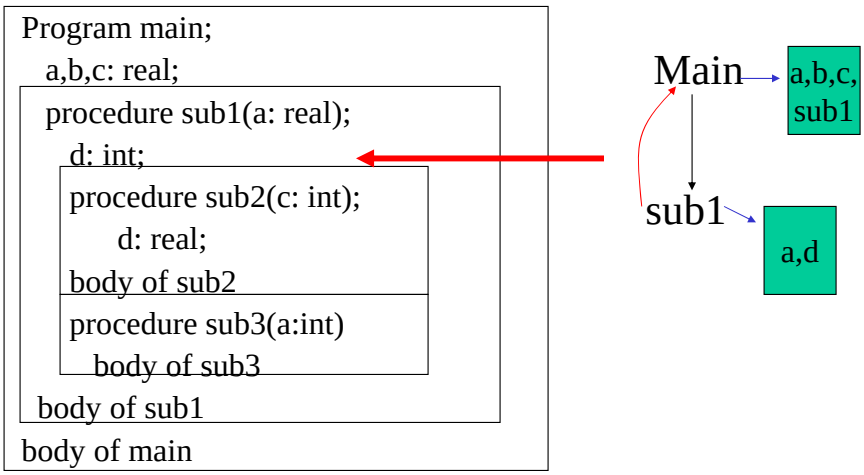
We can use a stack

- *EnterScope* – creates a new record that is a child of the current scope. This scope has new empty local table. Set CS to this record ➔ PUSH
- *ExitScope* – set CS to parent of current scope. Update tables ➔ POP
- *Insert* – add a new entry to the local table of CS
- *Lookup* – Search local table of CS. If not found, check the enclosing scope. Continue checking enclosing scopes until found or until run out of scopes.

Example Program – As we compile …

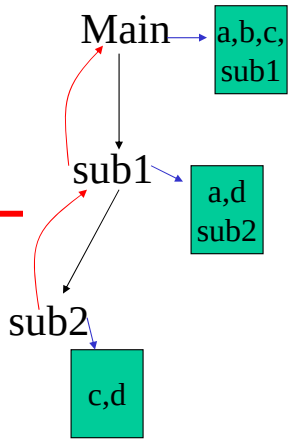


Example Program



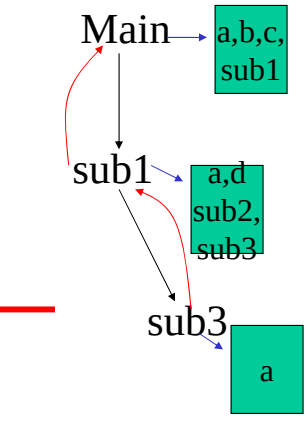
Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



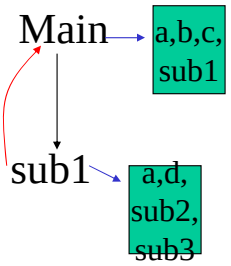
Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```



Example Program

```
Program main;
  a,b,c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
        d: real;
    body of sub2
    procedure sub3(a:int)
      body of sub3
  body of sub1
body of main
```