CSC 4181
Compiler Construction

Scope and Symbol Table

Scope

- A scope is a textual region of the program in which a (name-to-object) binding is active.
- There are two types of scope:
  - Static scope
  - Dynamic scope
- Most modern languages implement static scope (*i.e.*, the scope of binding is determined at compile-time).

Static Scope

- Static scope is also called lexical scope because the bindings between name and objects can be determined by examining the program text.
- Typically, the current binding for a given name is the one encountered most recently in a top-to-bottom scan of the program.

Static Scope Rules

- The simplest static scope rule has only a single, global scope (*e.g.*, early Basic).
- A more complex scope rule distinguishes between global and local variables (*e.g.*, Fortran).
- Languages that support nested functions (*e.g.*, Pascal, Algol) require an even more complicated scope rule.

Closest Nested Scope Rule

- A name that is introduced in a declaration is known
  - in the scope in which it is declared, and
  - in each internally nested scope,
  - unless it is hidden by another declaration of the same name in one or more nested scopes.

Nested subroutines in Pascal

```
procedure P1(A1 : T1);
var X : real;
    ...
    procedure P2(A2 : T2);
        ...
        procedure P3(A3 : T3);
        ...
        begin
            ...          (* body of P3 *)
        end;
        ...
    begin
        ...              (* body of P2 *)
    end;
    ...
    procedure P4(A4 : T4);
        ...
        function F1(A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...          (* body of F1 *)
        end;
        ...
    begin
        ...              (* body of P4 *)
    end;
    ...
begin
    ...                  (* body of P1 *)
end
```

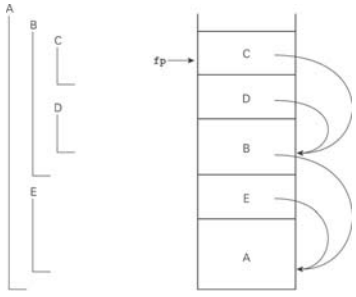
Hole and Qualifier

- A name-to-object binding that is hidden by a nested declaration of the same name is said to have a hole in its scope.
- In most languages, the object whose name is hidden is inaccessible in the nested scope.
- Some languages allow accesses to the outer meaning of a name by applying a qualifier or scope resolution operator.

Static Links and Static Chains

- A static link points to the activation record of its lexically-scoped parent.
- Static chain is a chain of static links connecting certain activation record instances in the stack.

Static Links and Static Chains



Scope in OOP

- In OOP, scope extends beyond functions and the main program.
- Each class defines a scope that cover every function's scope in that class.
- Inheritance and access modifiers also make some variables and functions visible outside their scope.

Dynamic Scope

- The bindings between names and objects depend on the flow of control at run time.
- In general, the flow of control cannot be predicted in advance by the compiler
- Languages with dynamic scoping tend to be interpreted rather than compiled.

Dynamic Links

- A dynamic link point to its caller activation record.

Static vs. Dynamic Scope

- Static scope rules match the reference (use of variable) to the closest lexically enclosing declaration.
- Dynamic scope rules choose the most recent active declaration at runtime.

Example: Static vs. Dynamic Scope

```
var a : integer;

procedure first
  a := 1;

procedure second
  var a : integer;
  first();

begin
  a := 2;
  second();
  write_integer(a);
end;
```

Static Scope: 1

Dynamic Scope: 2

Example: Static Scope

```
var a : integer;

procedure first
  a := 1;

procedure second
  var a : integer;
  first();

begin
  a := 2;
  second();
  write_integer(a);
end;
```

```
var a : integer;
main()
  a := 2;
  second()
    var a : integer;
    first()
      a := 1;
  write_integer(a);
```

The program prints 1

Example: Dynamic Scope

```
var a : integer;

procedure first
  a := 1;

procedure second
  var a : integer;
  first();

begin
  a := 2;
  second();
  write_integer(a);
end;
```

```
var a : integer;
main()
  a := 2;
  second()
    var a : integer;
    first()
      a := 1;
  write_integer(a);
```

The program prints 2

Referencing Environment

- A referencing environment is a set of active bindings at any point during program's execution.
- It corresponds to a sequence of scopes that can be examined in order to find the current binding for a given name.

Shallow and Deep Bindings

- When the referencing environment of a routine is not created until the routine is usually called, it is late binding.
- The late binding of the referencing environment is known as shallow binding.
- If the environment is bound at the time the reference is first created, it is early binding.
- The early binding of the referencing environment is called deep binding.

Example: Shallow vs. Deep Bindings (Dynamically Scoped Language)

```
var thres : integer;
function older(p : person) : boolean
  return p.age > thres
procedure show(p : person, c : function)
begin
  var thres : integer;
  thres := 20;
  if c(p)
    write(p)
end
procedure main(p)
begin
  thres := 35;
  show(p, older);
end
```

Deep binding: prints person p if older than 35

Shallow binding: prints person p if older than 20

Example: Deep Binding (Dynamically Scoped Language)

```
var thres : integer;
function older(p : person) : boolean
  return p.age > thres
procedure show(p : person, c : function)
begin
  var thres : integer;
  thres := 20;
  if c(p)
    write(p)
end
procedure main(p)
begin
  thres := 35;
  show(p, older);
end
```

```
main(p)
  thres := 35
  show(p, older)
    var thres : integer
    thres := 20
    older(p)
      return p.age > thres
    if <return value is true>
      write(p)
```


Example: Shallow Binding (Dynamically Scoped Language)

```
var thres : integer;
function older(p : person) : boolean
  return p.age > thres
procedure show(p : person, c : function)
begin
  var thres : integer;
  thres := 20;
  if c(p)
    write(p)
end
procedure main(p)
begin
  thres := 35;
  show(p, older);
end
```

```
main(p)
  thres := 35
  show(p, older)
    var thres : integer
    thres := 20
    older(p)
      return p.age > thres
    if <return value is true>
      write(p)
```

Shallow and Deep Bindings in Statically Scoped Language

- Shallow binding has never been implemented in any statically scoped language.
- Shallow bindings require more work by a compiler.
- Deep binding in a statically scoped languages is an obvious choice.

Example: Shallow vs. Deep Bindings (Statically Scoped Language)

```
var thres : integer;
function older(p : person) : boolean
  return p.age > thres
procedure show(p : person, c : function)
begin
  var thres : integer;
  thres := 20;
  if c(p)
    write(p)
end
procedure main(p)
begin
  thres := 35;
  show(p, older);
end
```

Shallow binding: Doesn't make sense

Symbol Table

- A symbol table is a dictionary that maps names to the information the compiler knows about them.
- It is used to keep track of the names in statically scoped program.
- Its most basic operations are *insert* (to put a new mapping) and *lookup* (to retrieve the binding information for a given name).

Symbol Table

- Static scope rules in most languages require that the referencing environment be different in different parts of the program.
- It is possible to implement a semantic analyzer such that new mappings are inserted at the beginning of the scope and removed at the end.

The Problems

- The straightforward approach to maintaining a referencing environment is not practical due to:
  - Nested scope: an inner binding must hide its outer binding.
  - Forward reference: names are sometimes used before they are declared.

Multilevel Symbol Table

- Most static scope rules can be handled by augmenting a simple symbol table to allow embedding symbol tables.
- When an inner scope is entered, the compiler executes the **`enter_scope`** operation.
- It executes the **`leave_scope`** operation when exits.

Lookup Operation

- When a lookup operation is initiated, the current symbol table is examined.
- If a given name is not found, an immediate outer symbol table is examined.
- This process is repeated until a binding is found for the name or an outermost symbol table is reached.