CENG 242 - Chapter 4: Binding and Scope

Burak Metehan Tunçel - May 2022

1 Bindings and Environment

The most important feature of high level languages: programmers able to give names to program entities (variable, constant, function, type, ...). These names are called ***identifiers***. They are declared once, used n times.

A ***binding*** is a fixed association between an identifier and an entity such as a value, variable, or procedure. A declaration produces one or more bindings. For binding:

- Scope of identifiers should be known: What is the block structure?, Which blocks the identifier is available?
- What will happen if we use same identifier name again "`C` forbids reuse of same identifier name in the same scope. Same name can be used in different nested blocks. The identifier inside hides the outside identifier".

```
double f, y;
int f() { // Error
  ...
}
double y; // Error
```

Code 1

```
double y;
int f() {
  double f; // OK
  int y; // OK
}
```

Code 2

*An **environment** (or name space) is a set of bindings occurrences that are accessible at a point in the program.* Each expression or command is interpreted in a particular environment, and all identifiers used in the expression or command must have bindings in that environment. It is possible that expressions and commands in different parts of the program will be interpreted in different environments.

```
struct Person { ... } x;
int f(int a) {
  double y;
  int x;
  ... ①
}

int main() {
   double a;
   ... ②
}
```

Code 3

O(①)={struct Person ↦ type, x ↦ int, f ↦ func, a ↦ int, y ↦ double}

O(②)={struct Person ↦ type, x ↦ struct Person, f ↦ func, a ↦ double, main ↦ func}

Usually at most one binding per identifier is allowed in any environment. An environment is then a partial mapping from identifiers to entities.

A ***bindable*** entity is one that may be bound to an identifier. Programming languages vary in the kinds of entity that are bindable:

- `C`'s bindable entities are types, variables, and function procedures.
- `JAVA`'s bindable entities are values, local variables, instance and class variables, methods, classes, and packages.

2 Scope

The ***scope*** of a declaration is the portion of the program text over which the declaration is effective. Similarly, the ***scope*** of a binding is the portion of the program text over which the binding applies.

In some early programming languages, the scope of each declaration was the whole program. In modern languages, the scope of each declaration is influenced by the program's syntactic structure, in particular the arrangement of blocks.

2.1 Block Structure

A ***block*** defines the scope of the identifiers declared inside (boundary of the definition validity). For variables, they also define the lifetime. Each programming language has its own forms of blocks:

- The blocks of a `C` program are block commands ({ ... }), function bodies, compilation units (source files), and the program as a whole.
- The blocks of a `JAVA` program are block commands (...), method bodies, class declarations, packages, and the program as a whole.
- The block of a `Haskell` program are '`let` *definitions* `in` *expression*' defines a block expression. Also '*expression* `where` *definitions*' defines a block expression. (the definitions have a local scope and not accessible outside of the expression).

Block structure of the language is defined by the organization of the blocks.

2.1.1 Monolithic Block Structure

In a language with ***monolithic block structure***, the only block is the whole program, so the scope of every declaration is the whole program. In other words, all declarations are global.

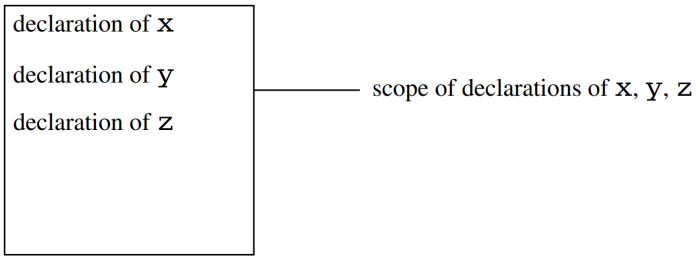

Figure 1: Monolithic block structure.

In a long program with many identifiers, they share the same scope and they need to be distinct.

2.1.2 Flat Block Structure

In a language with ***flat block structure***, the program is partitioned into several non-overlapping blocks. In other words, program contains *the global scope and only a single level local scope of function definitions*. No further nesting is possible.

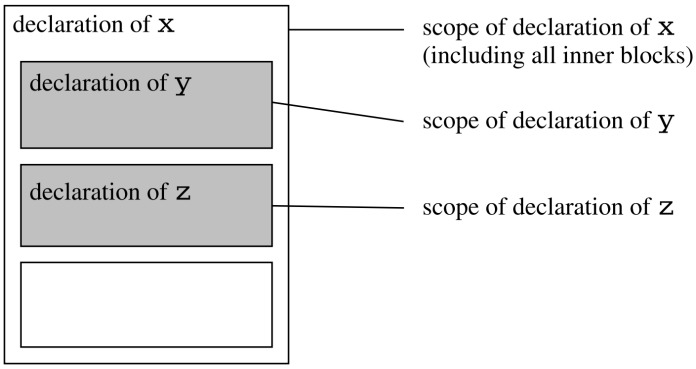

Figure 2: Flat block structure.

2.1.3 Nested Block Structure

In a language with ***nested block structure***, blocks may be nested within other blocks.

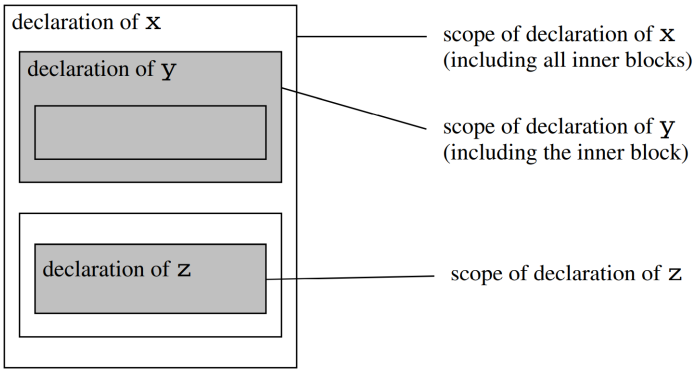

Figure 3: Nested block structure.

2.2 Scope and Visibility

Consider all the occurrences of identifiers in a program. We must distinguish two different kinds of identifier occurrences:

- A ***binding occurrence*** of identifier I is an occurrence where I is bound to some entity X.
- An ***applied occurrence*** of I is an occurrence where use is made of the entity X to which I has been bound. At each such applied occurrence we say that I denotes X.

When a program contains more than one block, it is possible for the same identifier I to be declared in different blocks. In general, I will denote a different entity in each block.

Consider two nested blocks, such that the inner block lies within the scope of a declaration of identifier I in the outer block:

- If the inner block *does not* contain a declaration of I, then applied occurrences of I both inside and outside the inner block correspond to the same declaration of I. The declaration of I is then said to be ***visible*** throughout the outer and inner blocks.
- If the inner block *does* contain a declaration of I, then all applied occurrences of I inside the inner block correspond to the inner, not the outer, declaration of I. The outer declaration of I is then said to be ***hidden*** by the inner declaration of I.

```
int x, y;

int f(double x) {
  ...              // parameter x hides global x
                   // in f()
}

int g(double a) {
  int y;          // local y hides global y in g()
  double f;       // local f hides global f()
                  // in g()
  ...
}

int main() {
  int y;         // local y hides global y
                 // in main()
}
```

Code 4

2.3 Static vs Dynamic Scoping

When are the binding and scope resolution done? In compile time or run time? Two options:

1. Static binding, static scope
2. Dynamic binding, dynamic scope

The first defines scope and binding based on the lexical structure of the program and binding is done *at compile time*. Second activates the definitions in a block during the execution of the block. The environment changes dynamically *at run time* as functions are called and returned.

2.3.1 Static Binding

Programs' shape is significant. Environment is based on the position in the source (lexical scope). Most languages apply static binding (`C, Haskell, Pascal, Java, ...`)

```
int x=1,y=2;
int f(int y) {
      y=x+y;      /* x global, y local */
      return x+y;
}
int g(int a) {
    int x=3;       /* x local, y global */
    y=x+x+a;    x=x+y;     y=f(x);
    return x;
}
int main() {
    int y=0;       /* x global y local */
    int a=10;
    x=a+y;     y=x+a;     a=f(a);     a=g(a);
    return 0;
}
```

Code 5

2.3.2 Dynamic Binding

Functions called update their declarations on the environment at run-time. Delete them on return. Current stack of activated blocks is significant in binding. `Lisp` and some script languages apply dynamic binding.

```
1  int x = 1, y = 2;
2
3  int f(int y) {
4    y = x + y;
5    return x + y;
6  }
7
8  int g(int a) {
9    int x = 3;
10   y = x + x + a; x = x + y;
11   y = f(x);
12   return x;
13 }
14
15 int main() {
16   int y = 0;  int a = 10;
17   x = a + y;  y = x + a;
18   a = f(a);   a = g(a);
19   return 0;
20 }
```

Code 6

	Trace	Environment (without functions)
	initial	{x:GL, y:GL }
12	call main	{x:GL, y:main, a:main }
15	call f(10)	{x:GL, y:f , a:main }
4	return f : 30	back to environment before f
15	in main	{x:GL, y:main, a:main }
15	call g(30)	{x:g, y:main, a:g }
9	call f(39)	{x:g, y:f, a:g }
4	return f : 117	back to environment before f
9	in g	{x:g, y:main, a:g }
10	return g : 39	back to environment before g
15	in main	{x:GL, y:main, a:main}
16	return main	x:GL=10, y:GL=2, y:main=117, a:main=39

3 Binding Process

Language processor keeps track of current environment in a data structure called ***Symbol Table*** or ***Identifier Table***. Symbol table maps identifier strings to their type and binding. Each new block introduces its declarations/bindings to the symbol table and on exit, they are cleared. Usually implemented as a *Hash Table*.

For static binding, Symbol Table is a compile time data structure and maintained during different stages of compilation. For dynamic binding, symbol table is maintained at run time.

4 Declaration

Definition: Creating a new name for an existing binding.

Declaration: Creating a completely new binding.

- in `C`: `struct Person` is a declaration. `typedef struct Person persontype` is a definition.
- in `C++`: `double x` is a declaration. `double &y=x;` is a definition.

The basic distinction is whether creating a new entity or not. However, usually the distinction is not clear and used interchangeably.

4.1 Type Declaration

A ***type declaration*** binds an identifier to a type. We can distinguish two kinds of type declaration. A ***type definition*** binds an identifier to an existing type. A ***new-type declaration*** binds an identifier to a new type that is not equivalent to any existing type.

4.2 Constant Declaration

A ***constant declaration*** binds an identifier to a constant value. A constant declaration typically has the form "`const I = E;`".

4.3 Variable Declaration

A ***variable declaration***, in its simplest form, creates a single variable and binds an identifier to that variable. A ***variable renaming definition*** binds an identifier to an *existing* variable. In other words, it creates an *alias*.

4.4 Procedure Definitions

A ***procedure definition*** binds an identifier to a procedure. In most programming languages, we can bind an identifier to either a function procedure or a proper procedure.

Note: The following subsections are important.

4.5 Sequential Declarations

A ***sequential declaration*** composes subdeclarations that are to be elaborated one after another. Each subdeclaration can use bindings produced by any *previous* subdeclarations, but not those produced by any *following* subdeclarations.

Declared identifier is not available in preceding declarations but is available in following declaration.

Most programming languages provide only such declarations.

4.6 Collateral Declarations

A ***collateral declaration*** composes subdeclarations that are to be elaborated independently of each other. These subdeclarations may not use bindings produced by each other. The collateral declaration merges the bindings produced by its subdeclarations.

Collateral declarations are uncommon in imperative and object-oriented languages, but they are common in functional and logic languages.

4.7 Recursive Declarations

A ***recursive declaration*** is one that uses the bindings that it produces itself. Such a construct is important because it enables us to define recursive types and procedures.

Declaration: `Name = Body`. The body of the declaration can access the declared identifier. Declaration is available in the body of itself.

`C` functions and type declarations are recursive. Variable definitions are usually not recursive.

4.8 Recursive Collateral Declarations

All declarations can access the others regardless of their order.

- All `Haskell` declarations are recursive collateral (including variables). All declarations are mutually recursive.
- `C++` class members are like this.
- In `C`, a similar functionality can be achieved by prototype definition.

4.9 Scopes of Declarations

Collateral, *sequential*, and *recursive declarations* differ in their influence on scope:

- In a *collateral declaration*, the scope of each subdeclaration extends from the end of the collateral declaration to the end of the enclosing block.
- In a *sequential declaration*, the scope of each subdeclaration extends from the end of that subdeclaration to the end of the enclosing block.
- In a *recursive declaration*, the scope of every subdeclaration extends from the start of the recursive declaration to the end of the enclosing block.

5 Blocks

If we allow a command to contain a local declaration, we have a *block command*. If we allow an expression to contain a local declaration, we have a*block expression*.

5.1 Block Commands

A block command is a form of command that contains a local declaration (or group of declarations) D and a subcommand C. The bindings produced by D are used only for executing C.

In other words, declarations done inside a block command is available only during the block. Statements inside work in this environment. The declarations lost outside of the block.

```
int x = 3, i = 2;
x += i;
while (x > i) {
  int i = 0;
  ...
  i++;
}
/* i is 2 again */
```

Code 7

5.2 Block Expressions

A ***block expression*** is a form of expression that contains a local declaration (or group of declarations) D and a subexpression C. The bindings produced by D are used only for evaluating E.

In other words, it allows an expression to be evaluated in a special local environment. Declarations done in the block is not available outside.

```
x=5
t=let xsquare=x*x
      factorial n = if n<2 then 1
                    else n*factorial (n-1)
      xfact = factorial x
  in  (xsquare+1)*xfact/(xfact*xsquare+2)
```

Code 8

Hiding works in block expressions as expected:

```
x=5 ; y=6 ; z = 3
t=let x=1
  in let y=2
     in   x+y+z
{--
t is 1+2+3 here.
local x and y hides the ones above
--}
```

Code 9

GCC (only GCC) block expressions has the last expression in block as the value:

```
double min ;
...
min =  ({ double tmp;
           if (b < a) then {
             tmp = a;  a = b ; b = tmp;
           }
           a; // this is the value of the block
         });
```

Code 10

5.3 Block Declaration

A declaration is made in a local environment of declarations. Local declarations are not made available to the outer environment.

In `Haskell`: $\texttt{D}_{exp}$ `where` $\texttt{D}_1$`;` $\texttt{D}_2$`; ... ;` $\texttt{D}_n$

Only $\texttt{D}_{exp}$ is added to environment. Body of $\texttt{D}_{exp}$ has all local declarations available in its environment.

```
fifthpower x = (forthpowerx) * x where
               squarex = x*x;
               forthpowerx = squarex*squarex
```

Code 11