Programming Language Concepts

Programming Language Theory

Topics

- Name and Binding
- Environments and Blocks
- **Scope Rules**

Scope Rules

- We already learned about visibility rules, which is also called ***scope rules***.
- These rules roughly, informally describe how names are visible in various environments regarding blocks.
- In this lecture, we will learn about scope rules in ***static and dynamic*** perspective.

Static vs. Dynamic

- ***Static scope*** (or lexical scope) depends solely on the syntactic structure of the program itself.
    - hence the environment can be determined completely by the compiler.
- ***Dynamic scope*** uses backward execution of the program to determine bindings.
    - hence it can be determined during runtime.

Static Scope Rule

- The static scope rule can be considered as ***the rule of nearest nested scope***.

- It is defined by the following three rules.

  - **Rule 1**: The declarations local to a block define the local environment of that block.

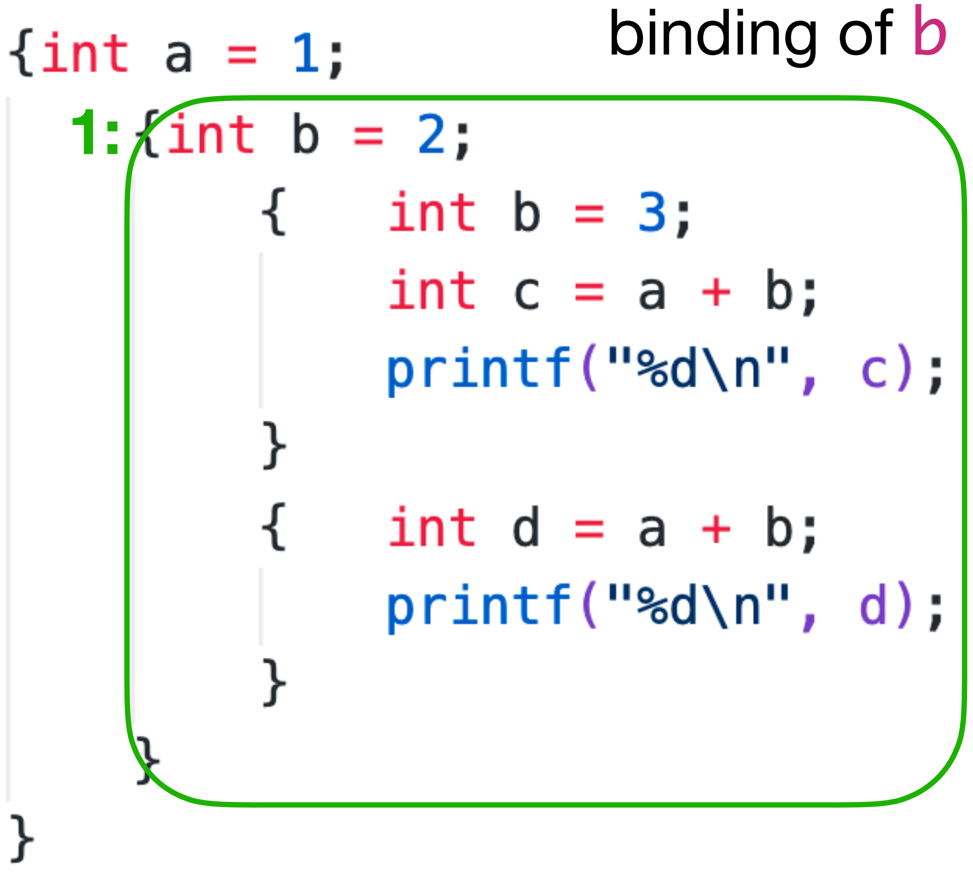
  - **Rule 2**: If a name is used inside a block, the valid binding of this name is the one presents in the local environment. If it doesn't exist, the one in the nearest outer block.

  - **Rule 3**: A block itself can be associated with names, and these names are part of the local environment of the block.

Rule 1: Local Declaration

- Locally declared variables define the local environment.
- In case of block 1, only variable b is declared in this block.
- Other variables are either not visible or visible, but not included in the local environment.

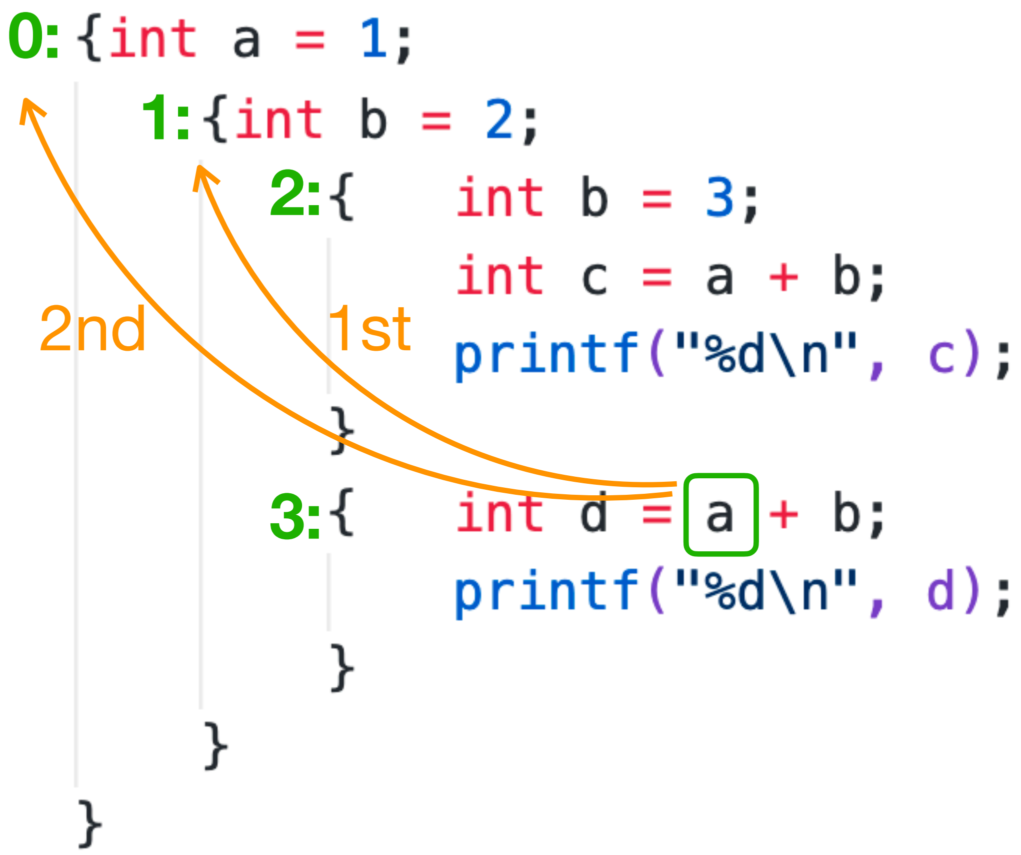
local environment of block 1
binding of b



```
{int a = 1;
 1:{int b = 2;
      {   int b = 3;
          int c = a + b;
          printf("%d\n", c);
      }
      {   int d = a + b;
          printf("%d\n", d);
      }
   }
}
```

Rule 2: Nearest Nested Scope

- Variable *a* is referenced in block 3.
- However, *a* is not declared in this block.
- Based on rule 2, we search for block 1 first.
- Still not found, hence try block 0 → *a* is declared here.
- Note that we skipped block 2, since it only searches for "nested" blocks.



```
0: {int a = 1;
     1:{int b = 2;
          2:{   int b = 3;
                int c = a + b;
                printf("%d\n", c);
            }
          3:{   int d = a + b;
                printf("%d\n", d);
            }
       }
   }
```

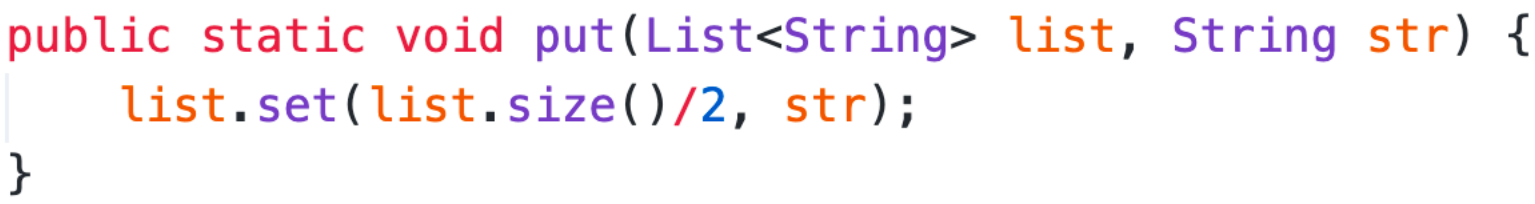
Rule 3: Names assigned to Block

- From the Java code, method name `put`, parameters `list` and `str` are not actually inside the block.

- However, they are available as the local environment.

- Also, they are not visible to outer blocks, since they are part of the local environment.

    - `put()` is an exception cause it's a procedure, which is visible to the block contains the declaration.



```
public static void put(List<String> list, String str) {
    list.set(list.size()/2, str);
}
```

Static Scope Advantages

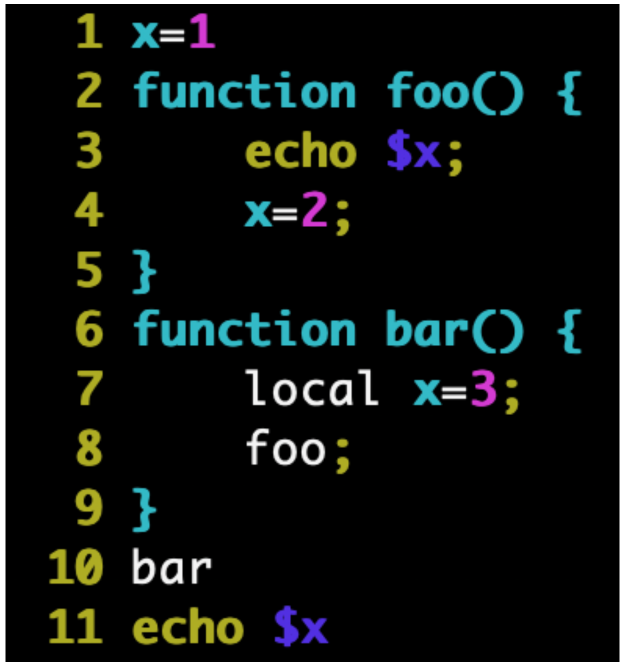
- All these static scope rules are pre-defined, and only depend on the syntactic structure of code.
- The compiler can deduce all the bindings of used names.
- This fact gives great advantages.
  - We can have better understanding of a program.
  - The compiler can perform correctness tests.
  - The compiler can perform considerable optimizations.

Dynamic Scope

- The valid binding of a name X at a certain point P of a program, is the most recent binding created for X.

- X must be still active at the point P.

Shell Script

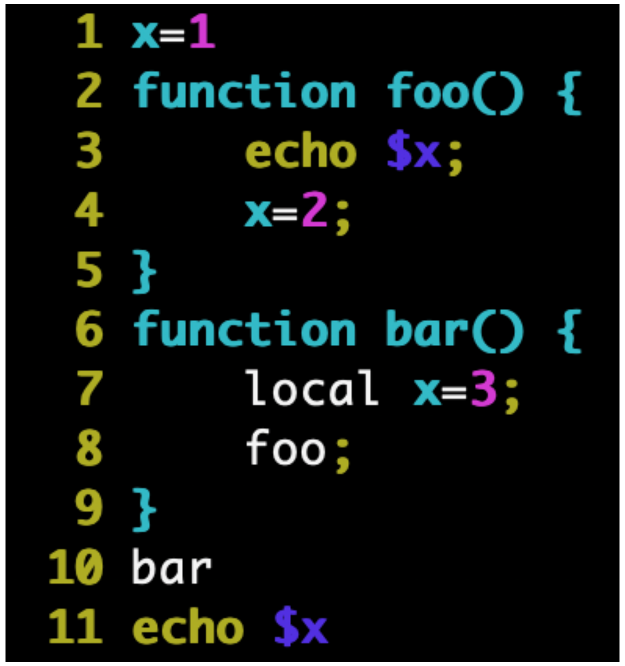


```
x=1
function foo() {
    echo $x;
    x=2;
}
function bar() {
    local x=3;
    foo;
}
bar
echo $x
```

Dynamic Scope

- If we consider the code on the right with static scope rules,
  - x at line 1 is a global variable.
  - Function bar is called at line 10.
  - It calls foo inside it.
  - Function foo prints x at line 3.
  - Then x is again printed at line 11.

Shell Script

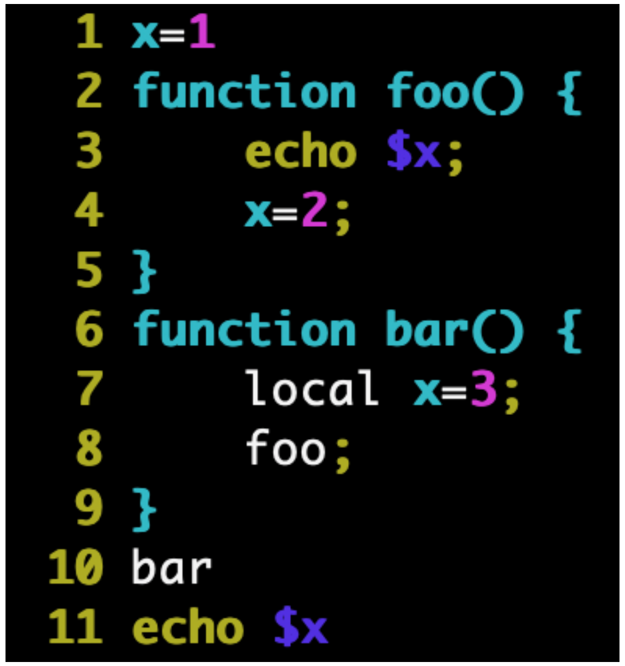


```
 1 x=1
 2 function foo() {
 3       echo $x;
 4       x=2;
 5 }
 6 function bar() {
 7       local x=3;
 8       foo;
 9 }
10 bar
11 echo $x
```

Dynamic Scope

- If we consider the code on the right with static scope rules,
    - x at line 1 is a global variable.
    - Function bar is called at line 10.
    - It calls foo inside it.
    - Function foo prints 1 at line 3 → using x at line 1.
    - Then x is again printed at line 11 → x is changed at line 4
        - So it prints 2.

Shell Script

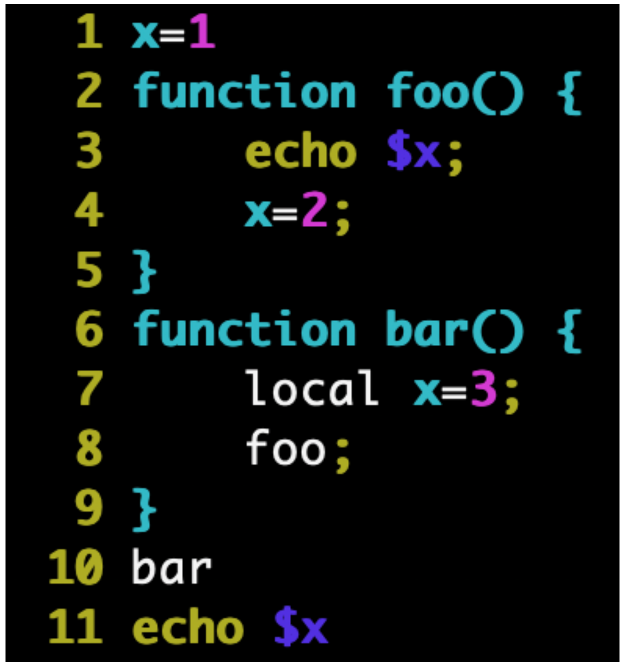


```
 1 x=1
 2 function foo() {
 3       echo $x;
 4       x=2;
 5 }
 6 function bar() {
 7       local x=3;
 8       foo;
 9 }
10 bar
11 echo $x
```

Dynamic Scope

- With dynamic scope, the real output of this script is,
  - **3** (printed by line 3)
    1 (printed by line 11)
- At line 3, the most recent binding of name x is at line 7.
  - Hence it prints 3.
- At line 11, the most recent binding of x to 2 at line 4 is already gone.
- So it prints 1.

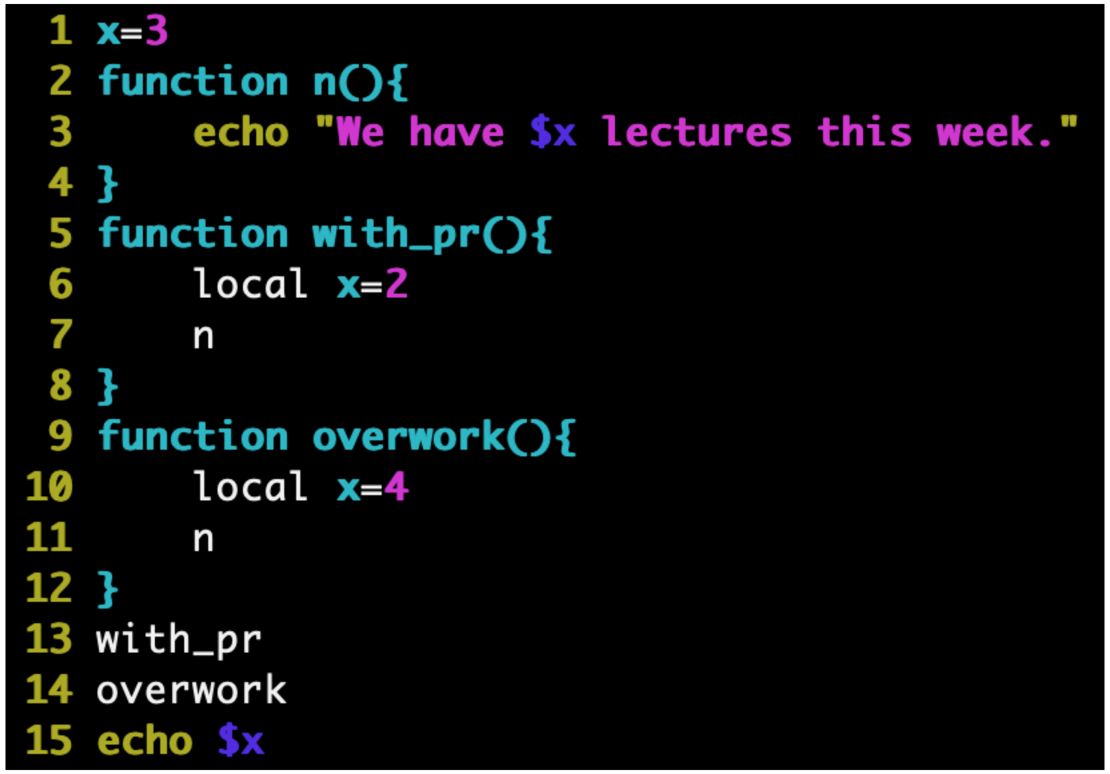
Shell Script



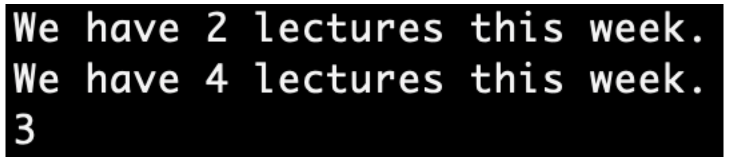
```
 1 x=1
 2 function foo() {
 3       echo $x;
 4       x=2;
 5 }
 6 function bar() {
 7       local x=3;
 8       foo;
 9 }
10 bar
11 echo $x
```

Dynamic Scope Advantages

- We can easily change the behaviour of functions *without parameters, and not modifying non-local variables.*
- Don't need to change the value of x.
- However, it makes difficult to understand the code easily.



```
x=3
function n(){
    echo "We have $x lectures this week."
}
function with_pr(){
    local x=2
    n
}
function overwork(){
    local x=4
    n
}
with_pr
overwork
echo $x
```



```
We have 2 lectures this week.
We have 4 lectures this week.
3
```

Summary

- Static Scope
- Dynamic Scope