Names, Bindings and Scopes

Programming Language Theory

Topics

- Names and Bindings
- Blocks and Environments
- Scope Rules

Names and Bindings

Name

- What is a **Name**?
- Name is merely a sequence of characters to represent (or denote) another object.
- In most programming languages, names have a form of identifiers.
  - e.g.) alphanumeric tokens (v1, v2, func, etc.) or sometimes other symbols (+, -).

Name ≠ Object

- A ***name*** and an ***object*** denoted by it ***are not the same thing***.
- One name can represent several different objects.
- Also, one object may have several different names.

Denotable Objects

- ***Denotable objects*** are the objects that we can give a name.
- Objects whose names are given ***by users***: variables, parameters, functions, user-defined types, etc.
- Objects whose names are given ***by PL***: primitive types, primitive operations, pre-defined constants.

Binding

- Association (or binding) between a name and an object it denotes can be created at various times.

- Although it is not theoretical, but practical, we use the terms “*static*” and “*dynamic*” for two principle phases.

- ***static***: Design of language, Program writing, Compile time

- ***dynamic***: Runtime

Language Design Bindings

- primitive types (int, double, etc), primitive operations (+, -, etc).
- Same thing can be denoted by different names in different languages.
  - Logical "AND" operations in Java and Python.

Java

```java
if(a > 0 && b > 0) {
    q = a / b;
}
```

Python

```python
if a > 0 and b > 0:
    q = a / b
```

Binding Times

- ***Program Writing***: programmers choose identifiers, which is a partial definition of bindings. Such bindings will be completed later.

- ***Compile Time***: while translating, a compiler allocates memory to some of the data structures, such as *global variables*.

- ***Runtime***: complete all bindings which have not been created yet.

  - e.g.) *local variables, pointer variables*, etc.

Referencing Environment

- ***Referencing Environment*** *(or simply* ***environment****)*, is a set of bindings between names and objects which exist at a specific point in the program at runtime.
- It is a set of bindings.
- For names and objects at a certain point of execution.
- Usually, we only refer to bindings not set up by language definition.

Declaration

- A ***Declaration*** is an introduction of a binding in the environment.
  - int x;
  - int func() {
        return 0;
    }
  - public class Foo;

Various Cases

- Bindings between names and denotable objects.
  - Single Name - Different Objects
  - Single Object - Different Names
    - In Different Environments
    - In the Same Environment - *aliasing.*

Single Name
Different Objects

- Here is an Example Java class.
- We have the same variable `sum` in two locations.
- Although their names are the same, they actually point to two different objects.

```
public class Example {
    public int sum; 1
    public int method() {
        int sum = 0; 2
        return sum;
    }
}
```

Single Object
Different Names

- In different environments, this is more common.
- Call by reference.
- Inside the method `put()`, a variable `list` denotes to the same `ArrayList`, `strings`.

```
public static void main() {
    List<String> strings = new ArrayList<>();
    put(strings, "Middle");
}
```

environment 1

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

This change will affect strings too.

environment 2

How about Call by Value?

- Call by Value copies the value and passes it to a method.
- So it is a case of different objects with a single or different names.
- Inside the method `put()`, a variable `oldStr` denotes to a different object, with the same value as `str` in `main()`.

```
public static void main() {
    String str = "Before";
    put(str, "After");
}
public static void put(String oldStr, String newStr) {
    oldStr = newStr;
}
```

This change will **not** affect str.

Single Object
Different Names

- In the same environments, this is more tricky.

- The case that a single object with different names is called ***aliasing***, and the different names are called ***aliases***.

- Consider the following C code snippet.

- What should be printed at the last line?

environment 1

```
Declare x, y              → int *x, *y;
Allocate heap memory      → x = (int *) malloc(sizeof(int));
* Dereference             → *x = 5;
y point to the same as x  → y = x;
                            *y = 10;
                            printf("%d\n", *x);
```

Blocks and Environments

Referencing Environment

- An ***environment*** is a set of bindings,
- for names and objects at a certain point of execution.
- Usually, we only refer to bindings not set up by language definition.

Blocks

- You may first think about `{ ... }`.
- A ***block*** is a textual region of a program, identified by a start and an end sign.
- A block can contain declarations local to that region.
    - i.e., such declarations are not valid outside that region.

Blocks

- Blocks can be represented in various ways.
- Usually, every time we enter and exit a block, the environment is changed.
- Blocks can be nested.
- Overlapping of blocks is never permitted.
  - i.e., we can't close a previous block, until the last opened block is closed.

Java

```java
if(a > 0 && b > 0) {
    q = a / b;
}
```

Python

```python
if a > 0 and b > 0:
    q = a / b
```

Scheme

```scheme
(define var "PL")
(let ((var 10))

)
```

Overlapping of Blocks

- Usually, inside a block, we will consider a local environment.
- Overlapping will make it very complicated.
- Hence this is not permitted in any PL.
- However, block nesting policy can be slightly different in PLs.

```java
public class Example {
    public int sum;
    public int method() {
        int sum = 0;
        return sum;
    }
}
```

Always matching like this.

```java
public class Example {
    public int sum;
    public int method() {
        int sum = 0;
        return sum;
    }
}
```

This is never allowed.

Types of Environment

- The environment associated with a block can be composed of the followings.
- ***Local environment*** is a set of bindings for names declared locally in the block.
- ***Non-local environment*** consists of bindings for names which are visible, but not declared in the block.
- ***Global environment*** is the environment from bindings created when the program begins.

Scope Rules

Visibility Rules

- This is somewhat informal concept.

- A ***local declaration*** in a block is ***visible*** to the block, and all the other blocks inside that block.

- If there is a new declaration of the same name in a block, this new declaration ***hides*** the previous one.

Visibility Rules

- There are block 0~3, and their ranges are represented by thin grey lines.
- When the current block is changed, the environment is changed.
- Hence the same name can be linked to a different object.
- What are the values of variables c and d?

```
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);
           }
       }
   }
```

Visibility Rules

- First of all, *a* is declared in block 0, hence it is visible to all blocks 0~3.
- The first b is declared in block 1, hence it is visible to blocks 1, 2, 3.
- The second b is declared in block 2, and visible to block 2 only.
- It also hides the first b in block 2, hence in block 2, b always denotes the second one.

```
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);
           }
       }
   }
```

Visibility Rules

- On the other hand, in block 3, the second b (in block 2) is not visible.
- Still, the first b in block 1 is visible to block 3, hence it is used to compute d.
- Therefore c is 4, and d is 3.

```
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);
           }
       }
   }
```

Environments

- Let's suppose that variable *a* is a global variable.
- Then *a* is visible to all blocks, and it is a part of the global environment.
- For block 1, the binding of *a* is ***global*** as well as ***non-local*** environment.

```
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);
           }
       }
   }
```

Environments

- Names in local environment is ***visible to inner blocks***.
- While names in local environment is ***not visible to outer blocks***.
- Names in non-local environment are ***hidden by the same name*** in local environment.
- More precisely, the binding of the first b *is deactivated* in local environment of block 2.

visible

not visible

```
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);
           }
       }
}
```

Still, this is not enough

- Visibility rules we discussed are roughly describing the big picture.
- Specific and detailed rules could be different in different PLs.
- For example, the case we just described is not valid in Java.

Java Example

- The previous example written in C, and works without errors.
- In Java, duplicate local variable is not allowed.
- On the other hand, we can still override a global variable.
- Therefore we have to understand specific rules for each programming language.

```
{int a = 1;
    {int b = 2;
        {   int b = 3;
            int c = a + b;
            System.out.println(c);
        }
        {   int d = a + b;
            System.out.println(d);
        }
    }
}
```

```
int b - Example1.main(String[])
Duplicate local variable b Java(536870967)
```

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 the 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);
          }
      }
  }
```

1st

2nd

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.

```java
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 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 (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.*
  - With runtime binding, we can decide a function's behaviour at runtime, not when we write the code.
- 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

- Names and Denotable Objects
- Bindings between Names and Objects
- Blocks and Environments
- Static vs. Dynamic Scope