Recitation 2: Scoping

Elaine Li (efl9013@nyu.edu)

Office Hours Fridays 1pm – 2pm, 418

Nisarg Patel(nisarg@nyu.edu)

Office Hours Mondays 11am – 12pm, 418

Introduction

- What is scoping?
- Static scoping
- Dynamic scoping
- Static scoping vs dynamic scoping examples
- Bonus: buddy system memory management

What is Scoping?

Terms:

- Name: character string used to represent something
  - E.g. variables, functions
- Binding: association between a name and what it represents
- Scope: region of program text where a binding is active
  - Where to find the value that is attached to a name
  - Where a name is valid in a program
  - What to do if the same name is defined twice

Example:

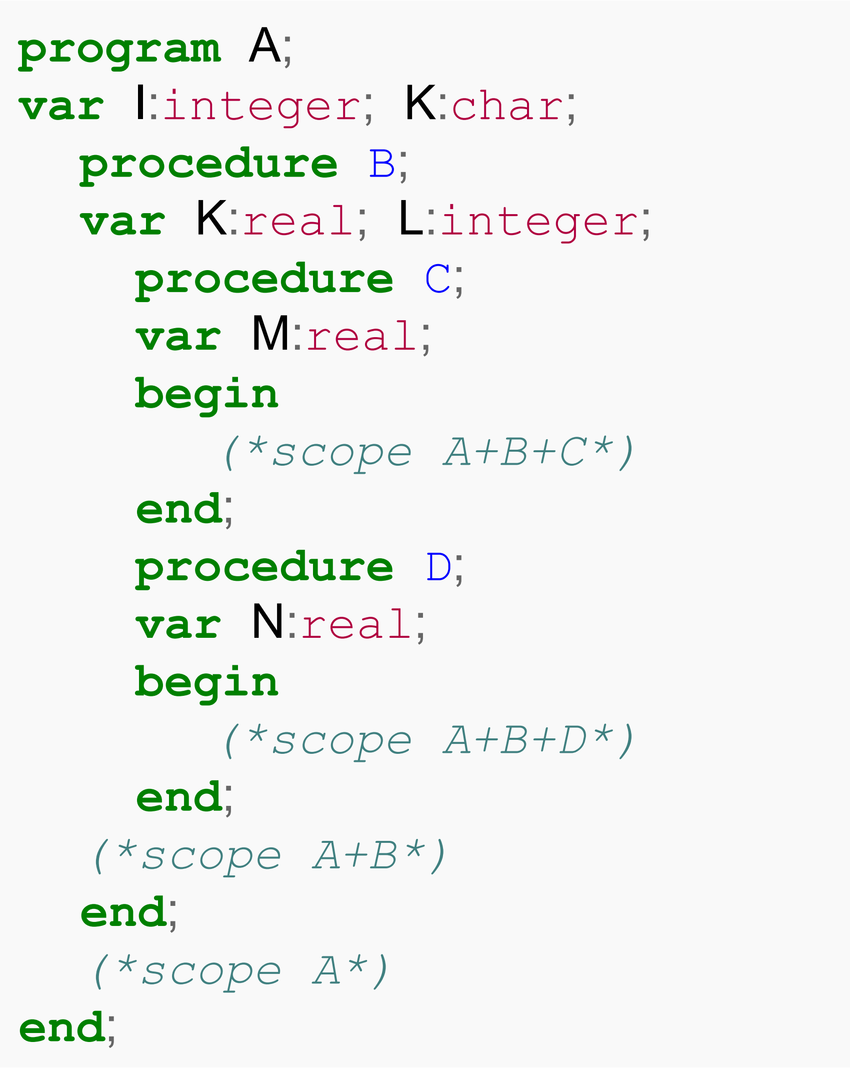
```
class Example1{
    private int globalVariable;
    public void foo(int arg1){
         int localVariable;
         …
    }
}
```

- What is the scope of:
  - globalVariable
  - 'foo'?
  - localVariable

Static Scoping

- The determination of bindings for names can be made by the compiler (early binding)
- All bindings for identifiers can be resolved by examining the program text
- Typically, we choose the most recent, active binding made at compile time
- Most modern languages use static scoping

Static Scoping Example:



```
program A;
var I:integer; K:char;
  procedure B;
  var K:real; L:integer;
    procedure C;
    var M:real;
    begin
       (*scope A+B+C*)
    end;
    procedure D;
    var N:real;
    begin
       (*scope A+B+D*)
    end;
   (*scope A+B*)
  end;
   (*scope A*)
end;
```

Dynamic Scoping

- The determination of bindings for names is made at runtime, and depends on control flow and execution order (late binding)
- Bindings for identifiers are resolved by looking at the “closest” binding on the program stack, and in general cannot be determined by the compiler
- Few languages implement dynamic scoping: Lisp, postscript

Dynamic Scoping

Advantages:

- Easier to implement in interpreters
- Programming convenience

Disadvantages:

- Hard to analyze routines with non-local variables that may be redefined by any caller of the routine
- Can lead to a lot of errors
- Binding errors may not be detected until run time

Static Scoping vs Dynamic Scoping Example 1

```c
int x;
int main() {
  x = 14;
  f();
  g();
}
void f() {
  int x = 13;
  h();
}
void g() {
  int x = 12;
  h();
}
void h() {
  printf("%d\n",x);
}
```

Static Scoping vs Dynamic Scoping Example 1

```
int x;
int main() {
  x = 14;
  f();
  g();
}
void f() {
  int x = 13;
  h();
}
void g() {
  int x = 12;
  h();
}
void h() {
  printf("%d\n",x);
}
```

Solution:
Static Scoping:
14
14

Static Scoping vs Dynamic Scoping Example 1

```
int x;
int main() {
  x = 14;
  f();
  g();
}
void f() {
  int x = 13;
  h();
}
void g() {
  int x = 12;
  h();
}
void h() {
  printf("%d\n",x);
}
```

Solution:
Static Scoping:
14
14

Dynamic Scoping:
13
12

Static Scoping vs Dynamic Scoping Example 2

```
int x;
int main() {
  x = 14;
  f();
  g();
}
void f() {
  x = 13;
  h();
}
void g() {
  x = 12;
  h();
}
void h() {
  printf("%d\n",x);
}
```

Static Scoping vs Dynamic Scoping Example 2

```
int x;
int main() {
  x = 14;
  f();
  g();
}
void f() {
  x = 13;
  h();
}
void g() {
  x = 12;
  h();
}
void h() {
  printf("%d\n",x);
}
```

Solution:

Static Scoping:

13

12

Static Scoping vs Dynamic Scoping Example 2

```
int x;
int main() {
  x = 14;
  f();
  g();
}
void f() {
  x = 13;
  h();
}
void g() {
  x = 12;
  h();
}
void h() {
  printf("%d\n",x);
}
```

Solution:

Static Scoping:

13

12

Dynamic Scoping:

13

12

Static Scoping vs Dynamic Scoping Example 3

```
const int b = 5;
int foo(){
   int a = b + 5;
   return a;
}
int bar(){
   int b = 2;
   return foo();
}
int main(){
   foo();
   bar();
   return 0;
}
```

Static Scoping vs Dynamic Scoping Example 3

```
const int b = 5;
int foo(){
   int a = b + 5;
   return a;
}
int bar(){
   int b = 2;
   return foo();
}
int main(){
   foo();
   bar();
   return 0;
}
```

Solution:

Static Scoping:

foo returns 10

bar returns 10

Static Scoping vs Dynamic Scoping Example 3

```
const int b = 5;
int foo(){
   int a = b + 5;
   return a;
}
int bar(){
   int b = 2;
   return foo();
}
int main(){
   foo();
   bar();
   return 0;
}
```

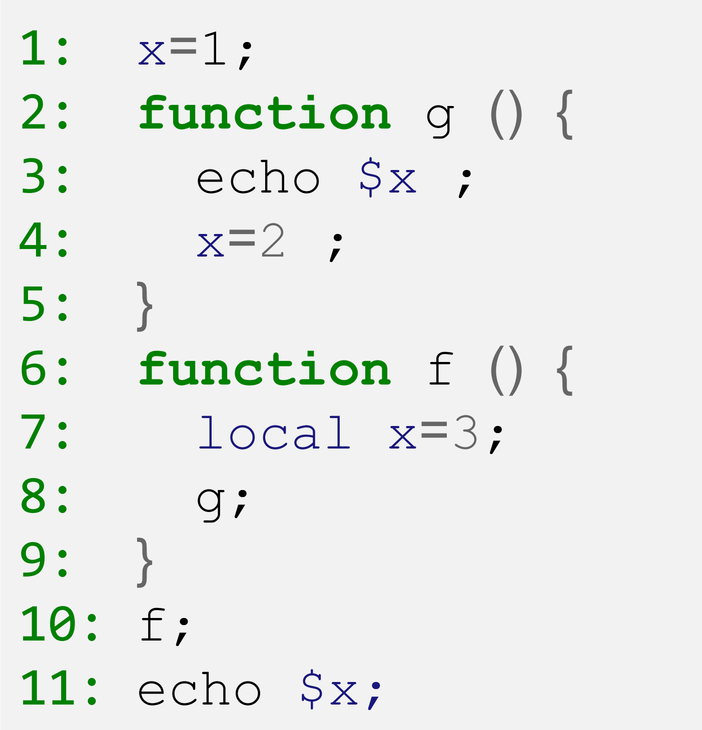
Solution:
Static Scoping:
foo returns 10
bar returns 10

Dynamic Scoping:
foo returns 10
bar returns 7

Static Scoping vs Dynamic Scoping Example 4

```
x=1;
function g () {
  echo $x ;
  x=2 ;
}
function f () {
  local x=3;
  g;
}
f;
echo $x;
```

Static Scoping vs Dynamic Scoping Example 4



```
x=1;
function g () {
  echo $x ;
  x=2 ;
}
function f () {
  local x=3;
  g;
}
f;
echo $x;
```

Solution:

Static Scoping:

1

2

Static Scoping vs Dynamic Scoping Example 4

```
x=1;
function g () {
  echo $x ;
  x=2 ;
}
function f () {
  local x=3;
  g;
}
f;
echo $x;
```

Solution:

Static Scoping:

1

2

Dynamic Scoping:

3

1

Static Scoping vs Dynamic Scoping Example 5

```
n:integer
procedure first
  n:=1
procedure second
  n:integer
  first()
n:=2
if read_integer() > 0
  second();
else
  first();
write_integer(n)
```

Static Scoping vs Dynamic Scoping Example 5

```
n:integer
procedure first
  n:=1
procedure second
  n:integer
  first()
n:=2
if read_integer() > 0
  second();
else
  first();
write_integer(n)
```

Solution:
Static Scoping:
1

Static Scoping vs Dynamic Scoping Example 5

```
n:integer
procedure first
  n:=1
procedure second
  n:integer
  first()
n:=2
if read_integer() > 0
  second();
else
  first();
write_integer(n)
```

Solution:
Static Scoping:
1
Dynamic Scoping:
It depends on the value given for read_integer(). If the input is positive then it is 2 otherwise 1

Static Scoping vs Dynamic Scoping Example 5

Explanation:

- Static scoping requires that the reference resolve to the closest lexically enclosed declaration. Procedure first changes n to 1 and write_integer prints the value.
- Dynamic scoping require that we choose the most recent binding for n at run time.
- We create a new binding for n when we enter the main program. Another binding for n is created in the procedure second. When we execute the assignment statement “n:=1”, the n to which we are referring will depend on whether we entered first through second or directly from the main program.
- If we entered procedure first through procedure second, value 1 will be assigned to second’s local n.
- If we entered procedure first through main program, we will assign the value 1 to the global n.
- In either case the line write_integer(n) will refer to the global variable n, since second’s n will be destroyed along with its binding, when the control returns to the main program.

Buddy system memory management

Key idea: store powers of two-size free blocks of memory in a binary tree

Advantages:

- Reduces external fragmentation
- Faster allocation and de-allocation

Disadvantages

- Does not avoid internal fragmentation (rounds to nearest power of 2)

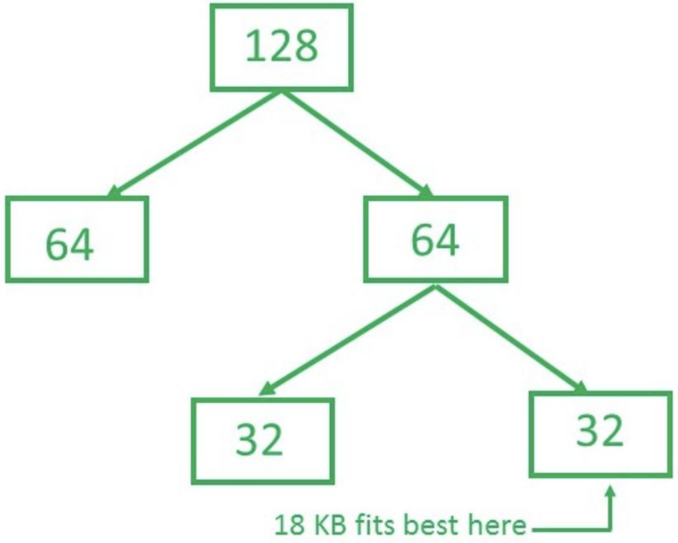


Image: https://www.geeksforgeeks.org/buddy-system-memory-allocation-technique/