22c:111 Programming Language Concepts

Fall 2008

Names

4.1 Syntactic Issues
4.2 Variables
4.3 Scope
4.4 Symbol Table
4.5 Resolving References
4.6 Dynamic Scoping
4.7 Visibility
4.8 Overloading
4.9 Lifetime

Recall that the term *binding* is an association between an entity (such as a variable) and a property (such as its value).

A binding is *static* if the association occurs before run-time.

A binding is *dynamic* if the association occurs at run-time.

Name bindings play a fundamental role.

The lifetime of a variable name refers to the time interval during which memory is allocated.

Syntactic Issues

Lexical rules for names.

Collection of reserved words or keywords.

Case sensitivity

- C-like: yes
- Early languages: no
- PHP: partly yes, partly no

Reserved Words

Cannot be used as *Identifiers*

Usually identify major constructs: *if while switch*

Predefined identifiers: e.g., library routines

Variables

Basic bindings

- Name
- Address
- Type
- Value
- Lifetime

L-value - use of a variable name to denote its address.

Ex: x = …

R-value - use of a variable name to denote its value.

Ex: … = … x …

Some languages support/require explicit dereferencing.

Ex: x := !y + 1

```
// Pointer example:
int x,y;
int *p;
x = *p;
*p = y;
```

Scope

The scope of a name is the collection of statements which can access the name binding.

In static scoping, a name is bound to a collection of statements according to its position in the source program.

Most modern languages use static (or *lexical*) scoping.

Two different scopes are either *nested* or *disjoint*.

In disjoint scopes, same name can be bound to different entities without interference.

What constitutes a scope?

	Algol	C	Java	Ada
Package	n/a	n/a	yes	yes
Class	n/a	n/a	nested	yes
Function	nested	yes	yes	nested
Block	nested	nested	nested	nested
For Loop	no	no	yes	automatic

The scope in which a name is defined or delared is called its *defining scope*.

A reference to a name is *nonlocal* if it occurs in a nested scope of the defining scope; otherwise, it is *local*.

```
1    void sort (float a[ ], int size) {
2    int i, j;
3    i = j = 0;
4    for (i = 0; i < size; i++)
5      for (j = i + 1; j < size; j++)
6            if (a[j] < a[i]) {
7                float t1,t2;
8                t1 = a[i];
9                a[i] = a[j];
10               a[j] = t2;
11           };
12     j = i + 1
13     };
14   i = 2*j }
```

```
1    void sort (float a[ ], int size) {  // C language
2    int i, j;
3    i = j = 0;
4    for (i = 0; i < size; i++)  // i, size local
5      for (j = i + 1; j < size; j++)  // j local
6            if (a[j] < a[i]) {  // a, i, j local
7                float t1,t2;
8                t1 = a[i];      // t1 local; a, i nonlocal
9                a[i] = a[j];    // j nonlocal
10               a[j] = t2;     // t2 local
11           };
12     j = i + 1                 // i, j local
13     };
14   i = 2*j }                   // i, j local
```

```
1    void sort (float a[ ], int size) {  // C++ language
2    int i, j;
3    i = j = 0;
4    for (int i = 0; i < size; i++) { // i local, size nonlocal
5      for (int j = i + 1; j < size; j++)  // j local
6            if (a[j] < a[i]) {  // j local,  a, i nonlocal
7                float t1,t2;
8                t1 = a[i];      // t1 local; a, i nonlocal
9                a[i] = a[j];    // j nonlocal
10               a[j] = t2;     // t2 local
11           };
12     j = i + 1;                // i local, j nonlocal
13     };
14   i = 2*j }                   // i, j local
```

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
    ...
}
... i ... // invalid reference to i
```

Symbol Table

A *symbol table* is a data structure kept by a translator that allows it to keep track of each declared name and its binding.

Assume for now that each name is unique within its local scope.

The data structure can be any implementation of a dictionary, where the name is the key.

1. Each time a scope is entered, push a new dictionary onto the stack.
2. Each time a scope is exited, pop a dictionary off the top of the stack.
3. For each name declared, generate an appropriate binding and enter the name-binding pair into the dictionary on the top of the stack.
4. Given a name reference, search the dictionary on top of the stack:
   a) *If found, return the binding.*
   b) *Otherwise, repeat the process on the next dictionary down in the stack.*
   c) *If the name is not found in any dictionary, report an error.*

```
1    void sort (float a[ ], int size) {  // C++ language
2    int i, j;
3    i = j = 0;
4    for (int i = 0; i < size; i++) { // i local, size nonlocal
5      for (int j = i + 1; j < size; j++)  // j local
6            if (a[j] < a[i]) {  // j local,  a, i nonlocal
7                float t1,t2;
8                t1 = a[i];      // t1 local; a, i nonlocal
9                a[i] = a[j];    // j nonlocal
10               a[j] = t2;     // t2 local
11           };
12     j = i + 1;                // i local, j nonlocal
13     };
14   i = 2*j }                   // i, j local
```

Example: previous C++ program

Bindings: ⟨var name, line location in code⟩

Dictionary Stack at line 7:

{⟨t1,7⟩, ⟨t2,7⟩}

{⟨j,4⟩}

{⟨i,4⟩}

{⟨a,1⟩, ⟨size,1⟩, ⟨i,2⟩, ⟨j,2⟩}

Resolving References

For static scoping, the *referencing environment* for a name is its defining scope and all nested subscopes.

The referencing environment defines the set of statements which can validly reference a name.

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15    int a, b;
16    h = 5; a = 3; b = 2;
17    A(a, b);
18    B(h);
19    ...
20 }
```

1. Outer scope: <h, 1> <i, 1> <B, 2> <A, 8> <main, 14>
2. Function B: <w, 2> <j, 3> <k, 4>
3. Function A: <x, 8> <y, 8> <i, 9> <j, 9>
4. Function main: <a, 15> <b, 15>

Symbol Table Stack for Function B:

<w, 2> <j, 3> <k, 4>

<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Symbol Table Stack for Function A:

<x, 8> <y, 8> <i, 9> <j, 9>

<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Symbol Table Stack for Function main:

<a, 15> <b, 15>

<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Line	Reference	Declaration
4	i	1
10	h	1
11	i	9
16	h	1
18	h	1

Dynamic Scoping

In dynamic scoping, a name is bound to its most recent declaration based on the program's call history.

Used be early Lisp, APL, Snobol, Perl.

Symbol table for each scope built at compile time, but managed at run time.

Scope pushed/popped on stack when entered/exited.

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6     ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
```

Using Figure 4.2 as an example: call history

main (17) → A (10) → B

Function	**Dictionary**
B	<w, 2> <j, 3> <k, 3>
A	<x, 8> <y, 8> <i, 9> <j, 9>
main	<a, 15> <b, 15>
	<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to <i, 9> in A.

```
1 int h, i;
2 void B(int w) {
3     int j, k;
4     i = 2*w;
5     w = w+1;
6         ...
7 }
8 void A (int x, int y) {
9     float i, j;
10    B(h);
11    i = 3;
12    ...
13 }
14 void main() {
15     int a, b;
16     h = 5; a = 3; b = 2;
17     A(a, b);
18     B(h);
19     ...
20 }
```

Using Figure 4.2 as an example: call history

main (17) → B

Function	**Dictionary**
B	<w, 2> <j, 3> <k, 3>
main	<a, 15> <b, 15>
	<h, 1> <i, 1> <B, 2> <A, 8> <main, 14>

Reference to i (4) resolves to <i, 1> in global scope.

Visibility

A name is *visible* if its referencing environment includes the reference and the name is not reclared in an inner scope.

A name redeclared in an inner scope effectively *hides* the outer declaration.

Some languages provide a mechanism for referencing a hidden name; e.g.: `this.x` in C++/Java.

```
1 public class Student {
2    private String name;
3    public Student (String name, ...) {
4       this.name = name;
5       ...
6    }
7 }
```

```
procedure Main is
    x : Integer;
    procedure p1 is
        x : Float;
        procedure p2 is
        begin
            ... x ...
        end p2;
    begin
        ... x ...
    end p1;
    procedure p3 is
    begin
        ... x ...
    end p3;
begin
    ... x ...
end Main; -- Ada
-- x in p2?
-- x in p1?  Main.x?
-- x in p3?  p1.x?
-- x in Main?
```

Overloading

Overloading uses the number or type of parameters to distinguish among identical function names or operators.

Examples:

- +, -, *, / can be float or int
- + can be float or int addition or string concatenation in Java
- `System.out.print(x)` in Java

Modula: library functions

- Read( ) for characters
- ReadReal( ) for floating point
- ReadInt( ) for integers
- ReadString( ) for strings

```
public class PrintStream extends
    FilterOutputStream {
    ...
    public void print(boolean b);
    public void print(char c);
    public void print(int i);
    public void print(long l);
    public void print(float f);
    public void print(double d);
    public void print(char[ ] s);
    public void print(String s);
    public void print(Object obj);
}
```

Lifetime

The *lifetime* of a variable is the time interval during which the variable has been allocated a block of memory.

Earliest languages used static allocation.

Algol introduced the notion that memory should be allocated/deallocated at scope entry/exit.

Remainder of section considers mechanisms which break *scope equals lifetime* rule.

C:

- Global compilation scope: static
- Explicitly declaring a variable `static`
- Remark: Java also allows a variable to be declared `static`