CMSC 124

DESIGN AND IMPLEMENTATION OF PROGRAMMING LANGUAGES

REFERENCING ENVIRONMENTS

BINDINGS AND SCOPING

BINDINGS

BINDING

Choosing a property from a set of possible properties for a particular program element.

BINDING TIME

Stage of program formulation when binding occurs.

The possible binding times are:

1.

Execution /Run Time

Bindings performed during program execution.

Commonly, **binding variables** to their **storage locations** and **values** occur at execution time.

Binding at run-time may occur during the following:

1. A.

UPON ENTERING A SUBPROGRAM/PROGRAM BLOCK

Example: Variable Declarations in C

```
int swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

a, b and temp are bound to their storage locations (addresses) when the function swap is called and their variable declarations are executed.

EXAMPLE: VARIABLE DECLARATIONS IN C

```
int swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

The values of **a** and **b** are bound upon entering the function; the parameters passed by the function call are assigned to them.

1. B.

AT ARBITRARY POINTS DURING EXECUTION

EXAMPLE: ASSIGNMENT STATEMENTS

```
int swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

The value of temp is bound when `= *a;` is executed.

EXAMPLE: ASSIGNMENT STATEMENTS

```
int swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

The value at the address contained by a (***a**) is bound (or re-bound) to the value at the address contained by b (***b**).

EXAMPLE: ASSIGNMENT STATEMENTS

```
int swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

The value at the address contained by b (**`*b`**) is bound (or re-bound) to the value of **`temp`**.

EXAMPLE: JAVA

```
void giffFunctionName(int a, int b) {
   int c;
   c = a + b;
   int d = c / 2 ;
}
```

Both **storage location** and **value binding** of **d** is done at an **arbitrary point of execution**.

2.

TRANSLATION/COMPILATION TIME

Bindings performed when the program is translated/compiled.

2. A.

BINDINGS CHOSEN BY THE PROGRAMMER

Variable names, data types, and program statement structures are all chosen by the programmer and are bound at translation.

2. B.

BINDINGS CHOSEN BY THE TRANSLATOR

Bindings chosen by the translator without input from the programmer.

EXAMPLE. SUBPROGRAMS

The relative storage location of subprograms is determined by the translator without the knowledge of the programmer.

Example. Arrays

Arrays and array descriptors are handled by the translator and may be different across different implementations.

2. C.

BINDINGS CHOSEN BY THE LOADER

Variable addresses are usually relative to the subprogram to which they belong.

```
swap
1 a
2 b
3 temp
```

```
pow
1 base
2 exp
3 result
```

When **subprograms** are **separately compiled**, they are **linked together** at **load time** to form a single executable program.

swap
1 a
2 b
3 temp

pow
1 base
2 exp
3 result

swap
1 a
2 b
3 temp
pow
4 base
5 exp
6 result

Actual addresses must be allocated within the computer in which the program will run, and can be done during load time.

3.

LANGUAGE IMPLEMENTATION TIME

Bindings performed when the language implementation (compiler/interpreter) is made.

Some aspects of a language's definition may be platform-specific.

EXAMPLE: MATH

The underlying number representation and arithmetic operations may be dictated by computer hardware.

4.

LANGUAGE DEFINITION TIME

Bindings performed when the language is defined/designed.

Programming language structure is bound at language definition time.

EXAMPLE

Given the assignment statement

```
x = x + 10
```

what bindings occur at what particular times?

EXAMPLE

```
x = x + 10
```

At language definition time, the set of possible data types `x` can take is defined.

EXAMPLE

```
int x = 4;
...
x = x + 10
```

At translation time, the actual data type of `x` may be determined due to an earlier variable declaration.

EXAMPLE

```
x = x + 10
```

At language implementation time, the set of possible values for `x` is determined by (1) the data type and (2) the platform for which the implementation is being made.

EXAMPLE

```
x = x + 10
```

During run-time, the actual value of x is determined through this assignment statement (and possibly others before and after).

EXAMPLE

```
x = x + 10
```

At language definition time, the representation of statements as a string of program text is determined.

EXAMPLE

`x = x + 10`

At language implementation time, the representation of the number literal `10` as a bit string is determined.

EXAMPLE

```
x = x + 10
```

At language definition time, the use of the operator symbols = and + for assignment and addition respectively is determined.

EXAMPLE

```
x = x + 10
```

At translation time, the decision of using either integer or complex/floating point addition is made.

BONUS

- Give at most three bindings and their respective binding times given the following statement:

```
int x = 70;
```

EARLY BINDING

Most bindings are done at translation time, empahasizing efficiency.

Example: FORTRAN, C

Late Binding

Many bindings can be done at during run-time, emphasizing flexibility.

Example: LISP

In cases where both efficiency and flexibility are equally important, the choice of binding time remains open.

Example: Ada

Language design only specifies the earliest time a binding can be done; the actual binding time is dependent on the language implementation.

STATIC BINDING

Occurs before run-time and does not change for the remainder of execution.

Example: Static variables

Dynamic Binding

Occurs or can be changed during run-time.

Example: Variables in Java

TYPE BINDING

Binding a variable to its data type.

Explicit Declarations

Statements that specify the data type of a variable.

Example:
```
int x;
float y;
String str;
```

IMPLICIT DECLARATIONS

Associate data types with variables through default conventions.

Example:
```
my $x = 10;
$x = “hello”;
$x = 3.14159
```

Dynamic Type Binding

A variable's data type is bound when it is assigned a value.

Example:
```
my $x = 10;
$x = "hello";
$x = 3.14159
```

TYPE INFERENCE

The data type of a variable is inferred from the data type of the literals in the expression.

In these cases, explicit data types in declarations are no longer required.

DISADVANTAGE

Compile-time type checking is difficult.

EXAMPLE

```
my $a = 5;

my $b = 4;

…

$b = “Hello”;

if($a == $b) {...}
```

Type mismatches are hard to detect because the PL attempts to change the data types of operands to match.

DISADVANTAGE

Since type checking needs to be done during run-time, program efficiency is hampered.

Dynamically-typed languages are often interpreted and statically-typed languages are often compiled.

STORAGE BINDING

The binding of a variable to its storage location.

ALLOCATION

Process of binding a variable to its storage location.

DEALLOCATION

Process of taking a bound storage location and placing back in the pool of available memory.

LIFETIME

The time during which a variable is bound to a storage location.

Variables are classified according to their lifetimes.

1.

STATIC VARIABLES

Bound to their storage location before execution and stay bound throughout the program.

ADVANTAGE

Easy implementation of global variables.

ADVANTAGE

Easy addressing, since addresses will not change; programs become more efficient.

ADVANTAGE

Variables can be history-sensitive.

HISTORY-SENSITIVE VARIABLES

Variables declared in subprograms that remember the values of earlier executions of the same subprogram.

DISADVANTAGE

Not flexible.

DISADVANTAGE

Recursive programs are not supported.

DISADVANTAGE

Storage cannot be shared between variables.

Example

```
void foo() {
  int a[100], i;
  for(i = 0; i < 100; i++)
    a[i] = (abs(rand())%100);
}
void boo() {
  int a[100], i;
  for(i = 0; i < 100; i++)
    a[i] = i;
}
```

Static variables are supported in C-based languages using the `static` keyword.

2.

STACK -DYNAMIC VARIABLES

Variables allocated when their declarations are elaborated. Their storage is statically-bound.

ELABORATION

storage allocation and binding process that occurs when execution reaches the declaration during run-time.

Run-time stack

space within which stack-dynamic variables are allocated.

Stack-dynamic variables are prominently used for local variables in subprograms.

Storage is allocated at elaboration and deallocated upon returning control of execution to the caller, thus supporting recursive subprograms.

Each active copy of a subprogram has its own version of its stack-dynamic variables.

ADVANTAGE

Memory can be shared for local variables of subprograms.

DISADVANTAGE

Run-time overhead of allocation and deallocation.

DISADVANTAGE

No support for history-sensitive variables.

C and C++ implement local variables as stack-dynamic.

Other attributes of stack-dynamic variables are still statically-bound if the variable is scalar (e.g., data types).

3.

Explicit Heap-Dynamic Variables

Nameless, abstract memory cells allocated and deallocated by explicit run-time instructions by the programmer.

They can only be accessed using pointers or other reference variables.

EXAMPLE

An example of explicit run-time instructions for de/allocation are `malloc` and `free` in C.

EXAMPLE

```
int *arr, len;
printf(“Enter array length: ”);
scanf(“%d”, &len);
arr = (int *) malloc(sizeof(int) *
      len);
```
Explicit allocation

…

```
free(arr);
```
Explicit deallocation

EXAMPLE

Java objects are all explicit-heap dynamic and are accessed using reference variables, but implicit garbage collection is employed for deallocation.

EXAMPLE

```
int yetAnotherExample() {
   LinkedList<String> names =
         new LinkedList<String>();
}
```

Explicit allocation

Explicit heap-dynamic variables are commonly used for **dynamic structures**. (e. g., structures that **grow** and **shrink** during **run-time**).

DISADVANTAGE

Use of pointers and reference variables may be difficult and/or confusing.

DISADVANTAGE

Cost of pointer dereferencing, and dynamic allocation/deallocation.

4.

IMPLICIT HEAP-DYNAMIC VARIABLES

Variables bound to heap storage only when they are assigned values.

All variable attributes (data type, value, etc.) are bound every time an assignment statement is done.

ADVANTAGE

Give the highest degree of flexibility.

DISADVANTAGE

Run-time overhead of variable attribute maintenance.

DISADVANTAGE

Loss of error-detection capabilities.

Examples of languages that use implicit heap-dynamic variables are APL and ALGOL 68.

SCOPING

SCOPE

Range of statements in which a variable is visible.

VISIBLE VARIABLES

Variables that can be referenced in that statement.

SCOPE RULES

Determine the association between a name occurrence and a variable.

LOCAL VARIABLES

Variables that are declared inside a subprogram or program block and can be accessed there.

NONLOCAL VARIABLES

Variables that can be accessed inside a subprogram or program block, but is not declared there.

EXAMPLE

```
int a = 3; //nonlocal to foo
void foo() {
    int b = 5; //local to foo
    printf(“%d %d\n”, a, b);
}
```

Global variables are just a subset of nonlocal variables.

STATIC SCOPING

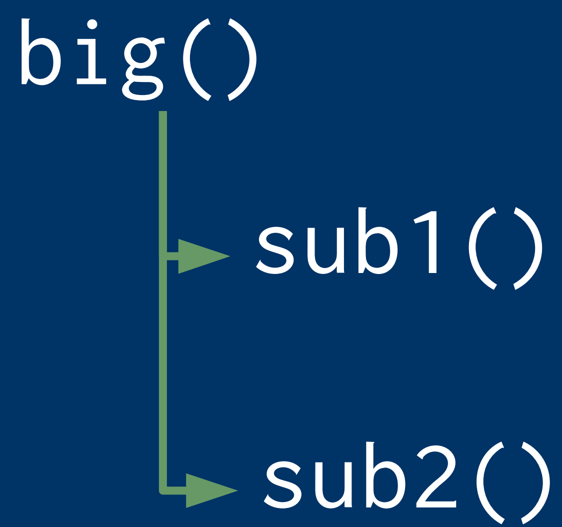
Scope of a variable can be determined before execution.

Used by many imperative languages.

Statically-scoped languages allow **nested subprograms**, **program blocks**, and **class definitions**.

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```



STATIC PARENT

The subprogram/program block in which a nested subprogram is declared.

Static Ancestors

The static parents, grandparents, etc., cascaded all the way to the largest enclosing subprogram/ program block.

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

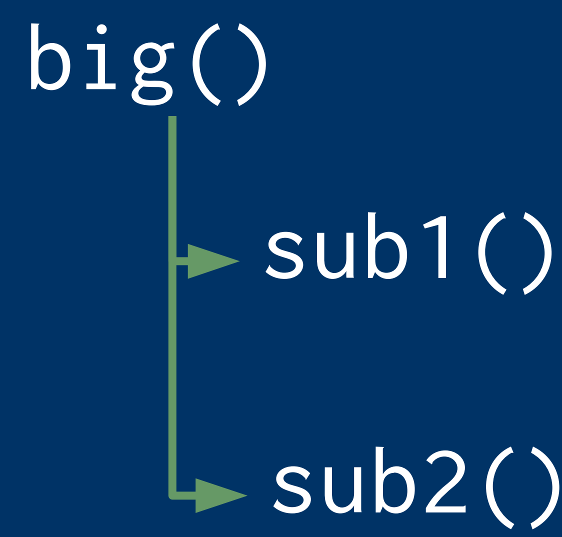
Sequence of calls:
`big()->sub1()->sub2()`

What is the **static parent** of...

- `sub1`?
- `sub2`?

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```



`big` is the static parent of both `sub1` and `sub2`.

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

Which value of **x** will **sub2** use? 3 or 7?

sub1()

BLOCK

Section of code where stack-dynamic variables are allocated upon entrance and deallocated upon exit.

BLOCK-STRUCTURED LANGUAGES

PLs that use blocks.

Example: C

```
if(list[i] < list[j]) {
   int temp;
   temp=list[i];
   list[i]=list[j];
   list[j]=temp;
}
```

Compound statements, like subprograms, can also be **nested**.

EXAMPLE: C

```
void sub() {
   int count;
   …
   while(...) {
      int count=0; count++;
   }
}
```

Inside the `while` loop, the `count` being used is the one declared inside the same `while` loop, NOT the `count` outisde the `while` loop.

EXAMPLE: C

```
void sub() {
  int count;
  …
  while(...) {
    int count=0; count++;
  }
}
```

The declaration of **count** inside the **while** blocks the **count** declared inside **sub**.

This is only legal in C and C++, but not in C# and Java.

Sometimes, the order of variable declarations is important.

C89/C90/ANSI C requires variable declarations to occur at the start of the block; C99 has no such restrictions.

EXAMPLE

```
int main() {
  printf(“Enter a number: ”);
  int x;
  scanf(“%d”, &x);
  printf(“The number you entered
          is %d.\n”, x);
}
```

Other C-based languages that allow variable declarations anywhere in the block include C++, Java, and C#.

Usually, the scope of a variable is from its declaration toward the end of the block.

GLOBAL SCOPE

Variables can be declared outside functions/subprograms, called global variables.

Local variables can have the same name as global variables, but languages have different ways of handling these name conflicts.

EXAMPLE: C

```
int x = 5;
int main() {
  int x = 7;
  printf("%d", x);
}
```

`main`'s **x** will take precedence over the global **x**.

EXAMPLE: PHP

```
<?php
$x = 75;
$y = 25;
function addition() {
  $GLOBALS['z']=$GLOBALS['x']+$GLOBALS['y'];
}
addition();
echo $z;
?>
```

Use of GLOBALS array to separate global variables from local variables.

The existence of global scope can lead to more variables than necessary.

DYNAMIC SCOPE

Scope is now based on the calling sequence of programs.

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

Sequence of calls:

`big()->sub1()->sub2()`

What is the dynamic parent of...

- `sub1`?
- `sub2`?

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

`big` is the dynamic parent of `sub1`.

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

`sub1` is the dynamic **parent** of `sub2`.

EXAMPLE

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

What value of **`x`** will **`sub2`** use?

DISADVANTAGE

References to a variable name may not always be to the same variable.

DISADVANTAGE

Some variable attributes cannot be determined statically.

DISADVANTAGE

Some variable attributes cannot be determined statically.

Subprograms are executed within the environment of all previously called subprograms.

DISADVANTAGE

There is no way to protect local variables from access by any other executing subprogram.

In general, programs that use **dynamic scoping** are **less reliable** than those that use static scoping.

DISADVANTAGE

Type checking of nonlocal variables cannot be done statically.

DISADVANTAGE

Dynamically-scoped languages are often less readable.

DISADVANTAGE

Access to nonlocal variables take longer.

Remember, the lifetime of a variable is different from its scope.

REFERENCING ENVIRONMENT

Collection of variables visible in a statement.

If **static scoping** is used, the referencing environment consists of **all local scope variables** and all **variables in ancestor scopes**.

EXAMPLE

```
void uselessFunction2(int x) {
    int z = 5;
    void nestedFunction1() {
        int y = 11;
        void nestedFunction2() {
            int a = 6, b = 7;
            …
        }
        …
        nestedFunction2();
    }
    nestedFunction1();
}
```

If dynamic scoping is used, then the referencing environment consists of all locally-declared variables and the variables in other active subprograms.

ACTIVE SUBPROGRAM

A subprogram that has begun execution but has not terminated yet.

EXAMPLE

```
void sub1() {
  int a, b;
}
void sub2() {
  int b, c;
  …
  sub1();
}
void main() {
  int c, d;
  sub2();
}
```

NAMED CONSTANTS

Variables that can only be bound to their values once.

EXAMPLE: C

```
void main() {
   const int length = 100;
   int intList[length];
   float floatList[length];
}
```

EXAMPLE: JAVA

```
void example() {
   final int length = 100;
   int intList[length];
   float floatList[length];
}
```

INITIALIZATION

Binding of values to variables before the code block in which it is declared executes.

Named constants must often be initialized.