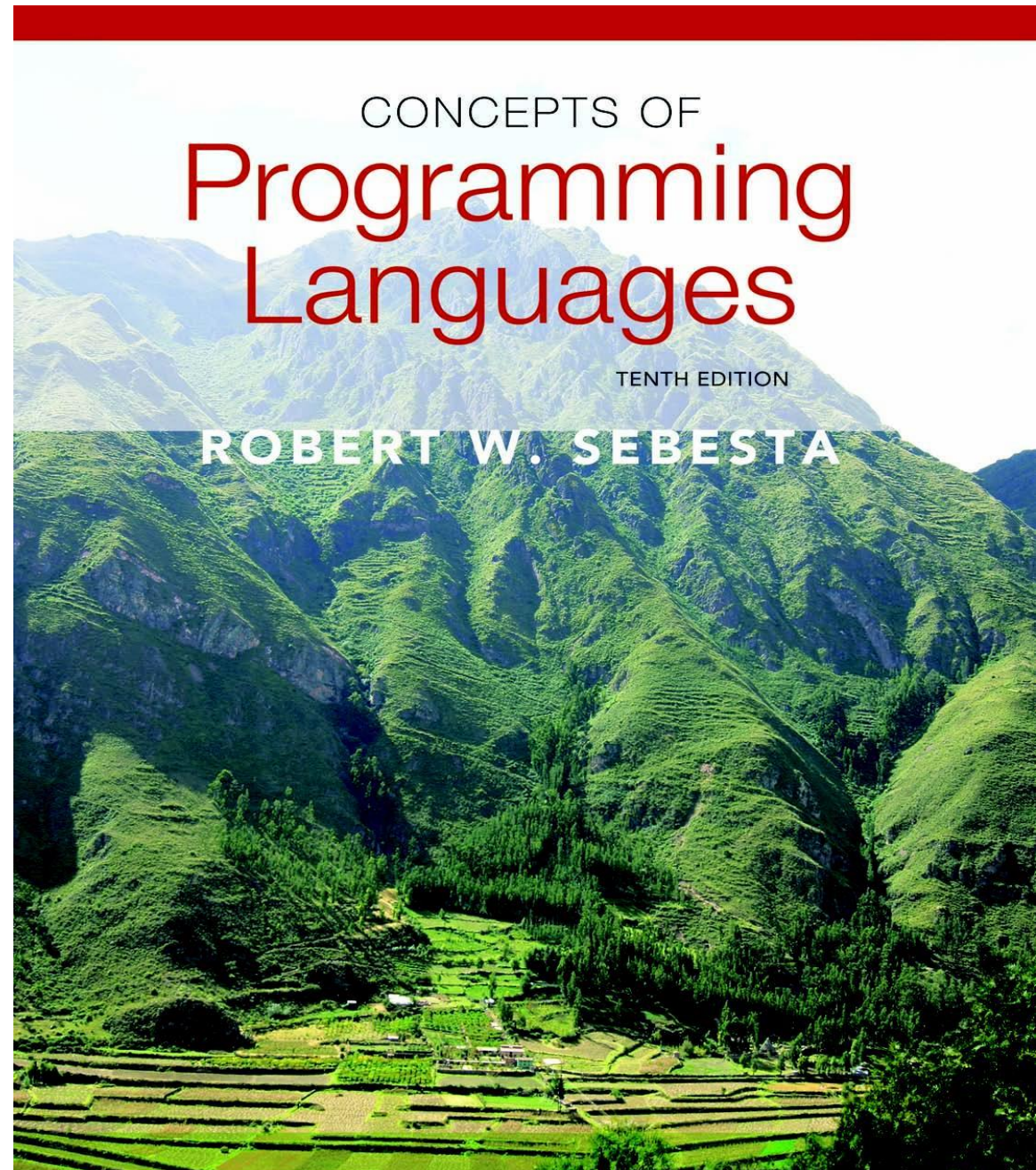


Chapter 5

Variables: Names, Bindings, and Scope

Chapter 5

Variables:
Names, Bindings,
Type Checking and
Scope



Introduction

This chapter introduces the fundamental semantic issues of **variables**.

- It covers the nature of names and special words in programming languages, attributes of variables, concepts of binding and binding times.
- It investigates type checking, strong typing and type compatibility rules.
- At the end it discusses named constraints and variable initialization techniques.

Names

Names

Design issues:

Maximum length?

Are connector characters allowed?

Are names case sensitive?

Are special words reserved words or keywords?

Length

FORTRAN I: maximum 6

COBOL: maximum 30

FORTRAN 90 and ANSI C: maximum 31

Ada: no limit, and all are significant

C++: no limit, but implementors often impose one

Connectors

Pascal, Modula-2, and FORTRAN 77 don't allow

Others do

Names (continued)

- **Special characters**
 - PHP: all variable names must begin with dollar signs
 - Perl: all variable names begin with special characters, which specify the variable's type
 - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Case sensitivity

- Foo = foo?
- The first languages only had upper case
- Case sensitivity was probably introduced by Unix and hence C.
- Disadvantage:
 - Poor readability, since names that look alike to a human are different; worse in Modula-2 because predefined names are mixed case (e.g. WriteCard)
- Advantages:
 - Larger namespace, ability to use case to signify classes of variables (e.g., make constants be in uppercase)
- C, C++, Java, and Modula-2 names are case sensitive but the names in many other languages are not

Special words

Def: A *keyword* is a word that is special only in certain contexts

- Disadvantage: poor readability
- Advantage: flexibility

Def: A *reserved word* is a special word that cannot be used as a user-defined name

Variables

- A *variable* is an abstraction of a memory cell
- Variables can be characterized as a 6-tuple of attributes:

Name: identifier

Address: memory location(s)

Value: particular value at a moment

Type: range of possible values

Lifetime: when the variable accessible

Scope: where in the program it can be accessed

Variables

- **Name** - not all variables have them (examples?)
- **Address** - the memory address with which it is associated
- A variable may have different addresses at different times during execution
- A variable may have different addresses at different places in a program
- If two variable names can be used to access the same memory location, they are called *aliases*
- Aliases are harmful to readability, but they are useful under certain circumstances

Aliases

- *How aliases can be created:*
 - Pointers, reference variables, Pascal variant records, C and C++ unions, and FORTRAN EQUIVALENCE (and through parameters - discussed in Chapter 8)
- Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN
 - replace them with dynamic allocation

Variables Type and Value

Type - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision

Value - the contents of the location with which the variable is associated

- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

lvalue and rvalue

Are the two occurrences of “a” in this expression the same?

a := a + 1;

In a sense,

- The one on the *left* of the assignment refers to the location of the variable whose name is a;
- The one on the *right* of the assignment refers to the value of the variable whose name is a;

We sometimes speak of a variable’s lvalue and rvalue

- The *lvalue* of a variable is its address
- The *rvalue* of a variable is its value

Binding

Def: A *binding* is an association, such as between an attribute and an entity, or between an operation and a symbol

Def: *Binding time* is the time at which a binding takes place.

Possible binding times:

- Language design time -- e.g., bind operator symbols to operations
- Language implementation time -- e.g., bind floating point type to a representation
- Compile time -- e.g., bind a variable to a type in C or Java
- Link time
- Load time--e.g., bind a FORTRAN 77 variable to memory cell (or a C static variable)
- Runtime -- e.g., bind a nonstatic local variable to a memory cell

Type Bindings

- *Def*: A binding is *static* if it occurs before run time and remains unchanged throughout program execution.
- *Def*: A binding is *dynamic* if it occurs during execution or can change during execution of the program.
- Type binding issues
 - How is a type specified?
 - When does the binding take place?
 - If static, type may be specified by either an explicit or an implicit declaration

Variable Declarations

Def: An *explicit declaration* is a program statement used for declaring the types of variables

Def: An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)

- E.g.: in Perl, variables of type scalar, array and hash begin with a \$, @ or %, respectively.
- E.g.: In Fortran, variables beginning with I-N are assumed to be of type integer.
- E.g.: ML (and other languages) use sophisticated type inference mechanisms
- Fortran, BASIC, Perl, Ruby, JavaScript, and PHP provide implicit declarations (Fortran has both explicit and implicit)

Advantages: writability, convenience

Disadvantages: reliability

Explicit/Implicit Declaration (continued)

- Some languages use type inferencing to determine types of variables (context)
 - C# – a variable can be declared with `var` and an initial value. The initial value sets the type
 - Visual BASIC 9.0+, ML, Haskell, F#, and Go use type inferencing. The context of the appearance of a variable determines its type

Dynamic Type Binding

- The type of a variable can change during the course of the program and, in general, is re-determined on every assignment.
- Usually associated with languages first implemented via an interpreter rather than a compiler.
- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited))
- Specified through an assignment statement, e.g. JavaScript

```
LIST = [2 4 6 8];
```

```
LIST = 17.3;
```

- *Advantages:*
 - Flexibility
 - Obviates the need for “polymorphic” types
 - Development of generic functions (e.g. sort)
- *Disadvantages:*
 - High cost (dynamic type checking and interpretation)
 - Type error detection by the compiler is difficult

Type Inferencing

- Type Inferencing is used in some programming languages, including ML, Miranda, and Haskell.
- Types are determined from the context of the reference, rather than just by assignment statement.

- Legal:

```
fun circumf(r) = 3.14159 * r * r; // infer r is real
```

```
fun time10(x) = 10 * x; // infer x is integer
```

- Illegal:

```
fun square(x) = x * x; // can't deduce anything
```

- Fixed

```
fun square(x) : int = x * x; // use explicit declaration
```

Storage Bindings and Lifetime

- Storage Bindings
 - *Allocation* - getting a cell from some pool of available cells
 - *Deallocation* - putting a cell back into the pool
- Def: The *lifetime* of a variable is the time during which it is bound to a particular memory cell
- Categories of variables by lifetimes
 - Static
 - Stack dynamic
 - Explicit heap dynamic
 - Implicit heap dynamic

Static Variables

- Static variables are bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.
- Examples:
 - all FORTRAN 77 variables
 - C static variables

Advantage: efficiency (direct addressing),
history-sensitive subprogram support

Disadvantage: lack of flexibility, no recursion!

Stack Dynamic Variables

- Stack-dynamic variables -- Storage bindings are created for variables when their declaration statements are elaborated.
(A declaration is elaborated when the executable code associated with it is executed)
- If scalar, all attributes except address are statically bound
 - e.g. local variables in Pascal and C subprograms (not declared static)
- *Advantages:*
 - allows recursion
 - conserves storage
- *Disadvantages:*
 - Overhead of allocation and deallocation
 - Subprograms cannot be history sensitive
 - Inefficient references (indirect addressing)

Static and Stack-Dynamic Variables

```
#include <stdio.h>

main() {
    int count = 0;    /* Count is a stack-dynamic variable */
    count = count + 1;
    printf("Sum is %d\n", sumValue(count)); /* sum = 1 */
    count = count + 1;
    printf("Sum is %d\n", sumValue(count)); /* sum = 3 */
    count = count + 1;
    printf("Sum is %d\n", sumValue(count)); /* sum = 6 */
}

int sumValue(int k)
{
    static int total = 0;    /* total is a static variable */
    total = total + k;
    return total;
}
```

Explicit heap-dynamic

Explicit heap-dynamic variables are allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution

- Referenced only through pointers or references
- e.g. dynamic objects in C++ (via `new` and `delete`), all objects in Java

Advantage: provides for dynamic storage management

Disadvantage: inefficient and unreliable

Example:

```
int *intnode;
```

```
...
```

```
intnode = new int;
```

```
...
```

```
delete intnode;
```

Implicit heap-dynamic

Implicit heap-dynamic variables -- Allocation and deallocation caused by assignment statements and types not determined until assignment.

e.g. all variables in APL; all strings and arrays in Perl, JavaScript, and PHP

Advantage:

- flexibility

Disadvantages:

- Inefficient, because all attributes are dynamic
- Loss of error detection

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The scope rules of a language determine how references to names are associated with variables

```
int x, y;  
int scope_function (int y) {  
    int x;  
    x = y + 3;    ← which x?  
}
```

- The *local variables* of a program unit are those that are declared in that unit
- The *nonlocal variables* of a program unit are those that are visible in the unit but not declared there
- *Global variables* are a special category of nonlocal variables

Variable Scope

- The *scope* of a variable is the range of statements in a program over which it's visible
- Typical cases:
 - Explicitly declared => local variables
 - Explicitly passed to a subprogram => parameters
 - The *nonlocal* variables of a program unit are those that are visible but not declared.
 - Global variables => visible everywhere.
- The scope rules of a language determine how references to names are associated with variables.
- The two major schemes are **static** scoping and **dynamic** scoping

Static Scope

- Aka “lexical scope”
- Based on program text and can be determined prior to execution (e.g., at compile time)
- To connect a name reference to a variable, you (or the compiler) must find the declaration
- *Search process*: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name
- Enclosing static scopes (to a specific scope) are called its *static ancestors*; the nearest static ancestor is called a *static parent*

Static Scoping

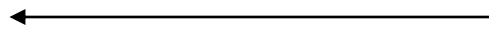
Example in Ada:

```
procedure Big is
  X: Integer;
  procedure Sub1 is
    begin    -- of Sub1
    ... X ...
    end;    -- of Sub1
  procedure Sub2 is
    X: Integer;
    begin    -- of Sub1
    ... X ...
    end;    -- of Sub1

begin    -- of Big
...
end;    -- of Big
```

First search for declaration of X in Sub1

If no declaration found, search continues to the next larger enclosing unit



Blocks

- A block is a section of code in which local variables are allocated/deallocated at the start/end of the block.
- Provides a method of creating static scopes inside program units
- Introduced by ALGOL 60 and found in most PLs.
- Variables can be hidden from a unit by having a "closer" variable with same name
- C++ and Ada allow access to these "hidden" variables

In Ada: unit.name

In C++: class_name::name

Examples of Blocks

C and C++:

```
for (...) {  
    int index;  
    ...  
}
```

Ada:

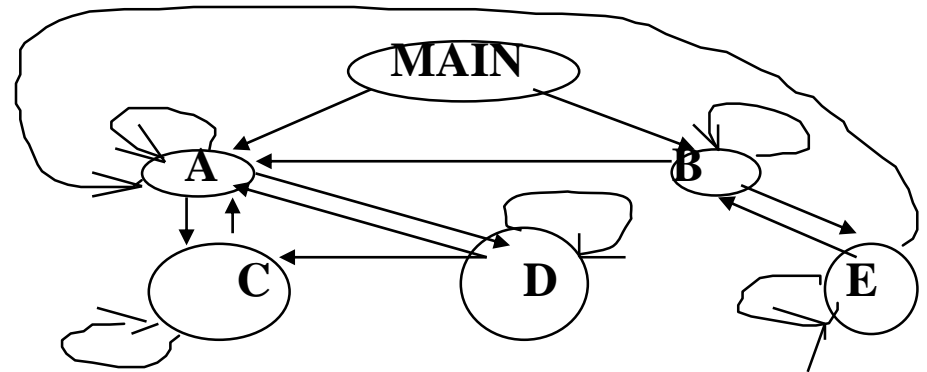
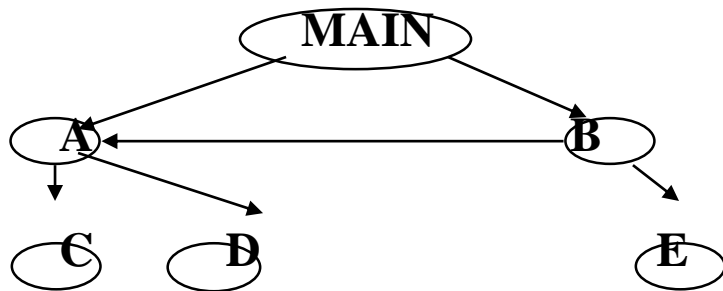
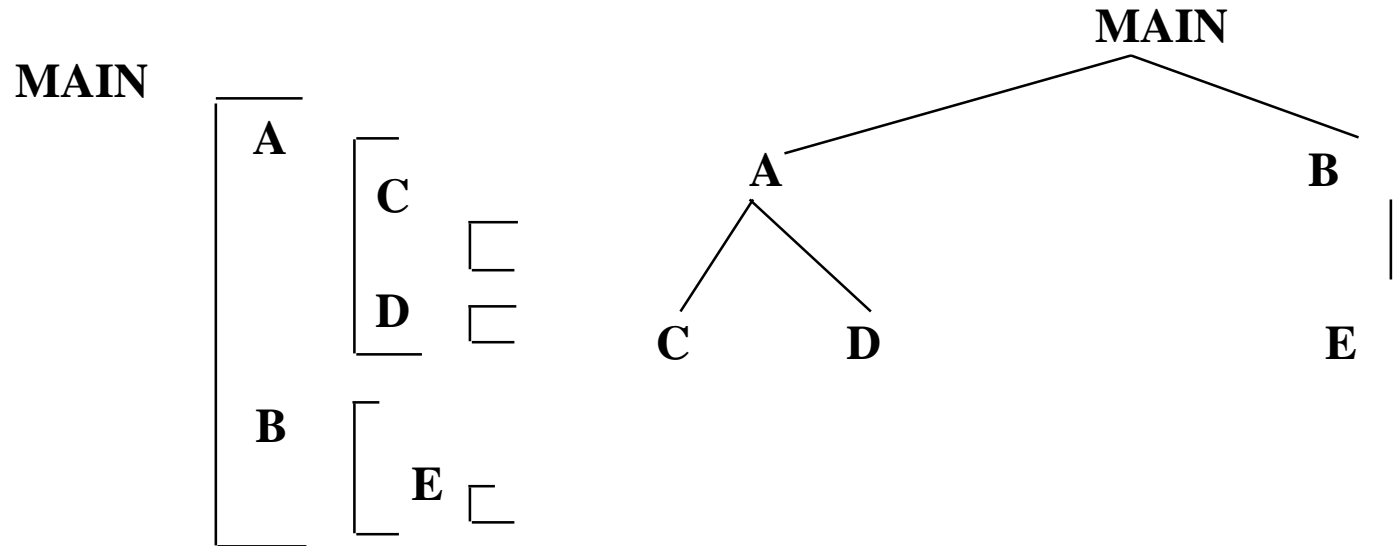
```
declare LCL : FLOAT;  
begin  
    ...  
end
```

Common Lisp:

```
(let ((a 1)  
      (b foo)  
      (c))  
    (setq a (* a a))  
    (bar a b c))
```

Static scoping example

MAIN calls A and B
 A calls C and D
 B calls A and E



Evaluation of Static Scoping

Suppose the spec is changed so that D must now access some data in B

Solutions:

1. Put D in B (but then C can no longer call it and D cannot access A's variables)
2. Move the data from B that D needs to MAIN (but then all procedures can access them)

Same problem for procedure access!

Overall: static scoping often encourages many globals

Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear
 - In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
 - In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block
 - However, a variable still must be declared before it can be used

The `LET` Construct

- Most functional languages include some form of `let` construct
- A `let` construct has two parts
 - The first part binds names to values
 - The second part uses the names defined in the first part
- In Scheme:

```
(LET (  
  (name1 expression1)  
  ...  
  (namen expressionn)  
)
```

The `LET` Construct (continued)

- In ML:

```
let
  val name1 = expression1
  ...
  val namen = expressionn
in
  expression
end;
```

- In F#:

- First part: `let left_side = expression`
- (left_side is either a name or a tuple pattern)
- All that follows is the second part

Declaration Order (continued)

- In C++, Java, and C#, variables can be declared in `for` statements
 - The scope of such variables is restricted to the `for` construct

Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
 - These languages allow variable declarations to appear outside function definitions
- C and C++ have both declarations (just attributes) and definitions (attributes and storage)
 - A declaration outside a function definition specifies that it is defined in another file

Global Scope (continued)

- PHP

- Programs are embedded in HTML markup documents, in any number of fragments, some statements and some function definitions
- The scope of a variable (implicitly) declared in a function is local to the function
- The scope of a variable implicitly declared outside functions is from the declaration to the end of the program, but skips over any intervening functions
 - Global variables can be accessed in a function through the `$GLOBALS` array or by declaring it `global`

Global Scope (continued)

- Python
 - A global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be `global` in the function

Evaluation of Static Scoping

- Works well in many situations
- Problems:
 - In most cases, too much access is possible
 - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

Dynamic Scope

- Based on calling sequences of program units, not their textual layout (temporal versus spatial)
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point
- Used in APL, Snobol and LISP
 - Note that these languages were all (initially) implemented as interpreters rather than compilers.
- Consensus is that PLs with dynamic scoping leads to programs which are difficult to read and maintain.
 - Lisp switch to using static scoping as it's default circa 1980, though dynamic scoping is still possible as an option.

Dynamic Scoping

- **Scope of variable is based on calling sequence of subprograms**

```
procedure Big is
  X: Integer;
  procedure Sub1 is
    begin      -- of Sub1
    ... X ...
    end;      -- of Sub1
  procedure Sub2 is
    X: Integer;
    begin      -- of Sub1
    ... X ...
    end;      -- of Sub1
begin        -- of Big
...
end;        -- of Big
```

Calling sequence:
Big calls Sub2
Sub2 calls Sub1

- X in Sub1 refers to the declaration in Sub2
- Sub2 is the dynamic parent of Sub1

Static vs. dynamic scope

```
Define MAIN
  declare x
  Define SUB1
    declare x
    ...
    call SUB2
    ...

  Define SUB2
    ...
    reference x
    ...
  ...
  call SUB1
  ...
```

```
MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x
```

- Static scoping - reference to x is to MAIN's x
- Dynamic scoping - reference to x is to SUB1's x

Dynamic Scoping

Evaluation of Dynamic Scoping:

- *Advantage:* convenience
- *Disadvantage:*
 1. While a subprogram is executing, its variables are visible to all subprograms it calls
 2. Impossible to statically type check
 3. Poor readability– it is not possible to statically determine the type of a variable

Scope vs. Lifetime

- While these two issues seem related, they can differ
- In Pascal, the scope of a local variable and the lifetime of a local variable seem the same
- In C/C++, a local variable in a function might be declared **static** but its lifetime extends over the **entire execution** of the program and therefore, even though it is inaccessible, it is still in memory

Referencing Environments

- The *referencing environment* of a statement is the collection of all names that are visible in the statement
- In a static scoped language, that is the local variables plus all of the visible variables in all of the enclosing scopes. See book example (p. 220)
- A subprogram is *active* if its execution has begun but has not yet terminated
- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms. See book example (p. 221)

Referencing Environment (static scoping)

(Example in Ada)

```
procedure Example is  
  A, B: Integer;
```

```
  ...
```

```
  procedure Sub1 is
```

```
    X, Y: Integer;
```

```
    begin          -- of Sub1
```

```
    ...
```

```
    end;          -- of Sub1
```

```
  procedure Sub2 is
```

```
    X: Integer;
```

```
    procedure Sub3 is
```

```
      X: Integer;
```

```
      begin          -- of Sub3
```

```
      ...
```

```
      end            -- of Sub3
```

```
    begin          -- of Sub2
```

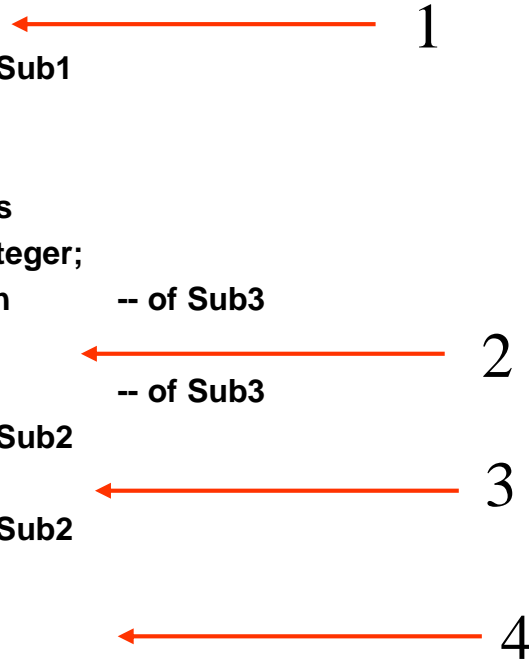
```
    ...
```

```
    end            -- of Sub2
```

```
  begin          -- of Example
```

```
  ...
```

```
  end            -- of Example
```



Referencing Environment:

- 1: X and Y of Sub1, A and B of Example
- 2: X of Sub3, (X of Sub2 is hidden), A and B of Example
- 3: X of Sub2, A and B of Example
- 4: A and B of Example

Referencing Environments (assuming dynamic scoping)

```
void sub1() {  
    int a, b;  
    ... ← 1  
} /* end of sub1 */
```

```
void sub2() {  
    int b, c;  
    ... ← 2  
    sub1  
} /* end of sub2 */
```

```
void main() {  
    int c, d;  
    ... ← 3  
    sub2();  
} /* end of main */
```

Referencing Environment:

1: a and b of sub1, c of sub2, d of main, (c of main and b of sub2 are hidden)

2: b and c of sub2, d of main, (c of main is hidden)

3: c and d of main

Named Constants

- A *named constant* is a variable that is bound to a value only when it is bound to storage.
- The value of a named constant can't be changed while the program is running.
- The binding of values to named constants can be either static (called manifest constants) or dynamic
- *Languages:*
 - Pascal:* literals only
 - Modula-2 and FORTRAN 90:* constant-valued expressions
 - Ada, C++, and Java:* expressions of any kind
- *Advantages:* increased readability and modifiability without loss of efficiency

Example in Pascal

```
Procedure example;  
  type a1[1..100] of integer;  
        a2[1..100] of real;  
  ...  
begin  
  ...  
  for I := 1 to 100 do  
    begin ... end;  
  ...  
  for j := 1 to 100 do  
    begin ... end;  
  ...  
  avg = sum div 100;  
  ...
```

```
Procedure example;  
  type const MAX 100;  
        a1[1..MAX] of integer;  
        a2[1..MAX] of real;  
  ...  
begin  
  ...  
  for I := 1 to MAX do  
    begin ... end;  
  ...  
  for j := 1 to MAX do  
    begin ... end;  
  ...  
  avg = sum div MAX;  
  ...
```

Variable Initialization

- For convenience, variable initialization can occur prior to execution
- FORTRAN: Integer Sum
Data Sum /0/
- Ada: Sum : Integer :=0;
- ALGOL 68: int first := 10;
- Java: int num = 5;
- LISP (Let (x y (z 10) (sum 0)) ...)

Summary

- Case sensitivity and the relationship of names to special words represent design issues of names
- Variables are characterized by the sextuples: name, address, value, type, lifetime, scope
- Binding is the association of attributes with program entities
- Scalar variables are categorized as: static, stack dynamic, explicit heap dynamic, implicit heap dynamic
- Strong typing means detecting all type errors

Summary

In this chapter, we see the following concepts being described

- Variable Naming, Aliases
- Binding and Lifetimes
- Scoping
- Referencing environments
- Named Constants