NAMES, BINDINGS AND SCOPE

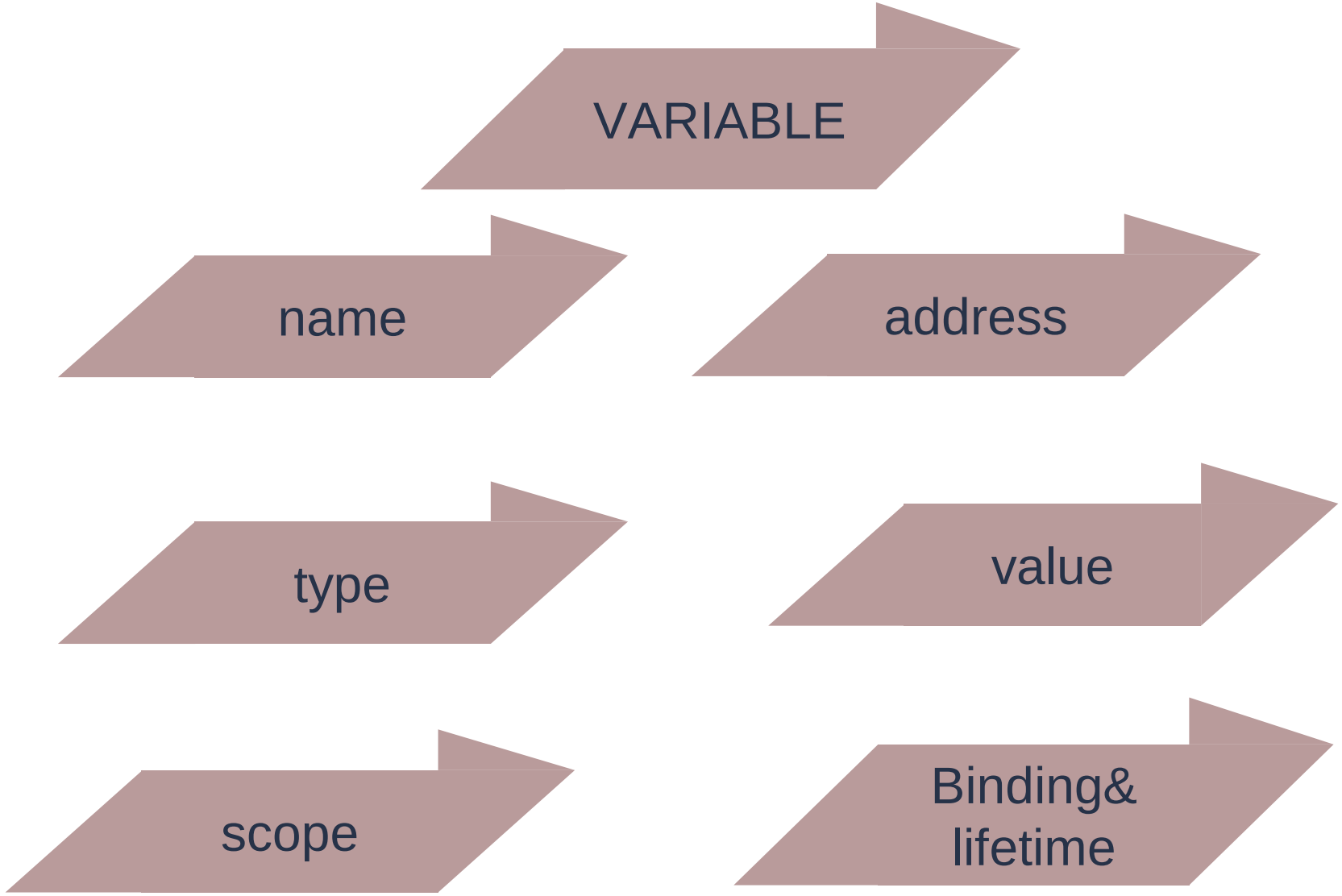
- Name: **name** is a string of characters used to identify some entity. Allow us to refer to variables, constants, functions, types, operations etc
- Binding: An association of a name with an object
- Scope: The part of the program in which the binding is active

Variables

- A variable in an imperative language, or an object-oriented language, is a six-tuple:

<name, address, value, type, lifetime, scope>

- A **name** is a string of characters used to identify some entity.
- Declaration of type, usage with a value, lifetime, scope of names are a major consideration in programming languages


VARIABLE
name
address
type
value
scope
Binding&
lifetime

For example, if we write **int x;**

- what will be the name of the variable and type of x?.
- The place of this declaration in the program decides where and how long x is available (scope, lifetime).
- Its address is determined when its program unit is executing
- Lastly , usage of x in statements decides what is its current value

What needs a name?

- constants, variables , operators,
- statement labels,
- procedures, functions, methods ,modules, programs,
- files, disks,
- commands, menu items,
- computers, networks, users

Name

- Name is a mnemonic character string representing something
- Names are identifiers(alpha numeric tokens)
- Names refer to abstraction-programmer associates name with complicated program fragment/element
  - Control abstraction
    - Allows the programmer to hide sequence of complicated code in a name.
  - Data Abstraction
    - Allows the programmer to hide data representation behind a set of operations.

- Names in most programming languages have the same form:
  a letter followed by a string consisting of letters, digits, and (_)
- use of the _ was widely used in the 70s and 80s(not popular)
- C-based languages (C, C++, Java, and C#), replaced the _ by the "camel" notation (ex: **myStack)**
- In C every keywords written in small letters
- In C, C++,Java etc. names are case sensitive

  Ex) rose, Rose, ROSE are distinct names

- the same name can be reused in different contexts and denote different entities.
- Different types, addresses and values may be associated with such occurrences of a name.
- Each occurrence has a different lifetime (when and for how long is an entity created?), and a different scope (where can the name be used?).

```
void a()
{
  int b;
 /* ... */
}
float b()
{
char a; /* ... */
}
```

- **Variable**- variable is an abstraction of a memory cell(s).
- **Name**

  - Not all variables have names: Anonymous, heap-dynamic variables

- **Address**

  • The memory address with which it is associated

  • A **variable name may have different addresses at different places** and at different times during execution

  - The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement

- **Type**

• Determines the **range of values of variables and the set of operations** that are defined for values of that type

- For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations (+,-,*,/,%)
- **Value**

• The value of a variable is the **contents of the memory cell** or cells associated with the variable.

• A variable's value is sometimes called its r-value because that is what is required when a variable appears in the right side of an assignment statement.

BINDING

Binding , Binding time,& Referencing Envt

- **Binding** is association of 2 things-an attribute with an entity.
- **Binding time** is the time at which a binding takes place.
- **Referencing Environment**-complete set of bindings at a given point in a program.

Binding

- ***Binding***
- the operation of associating two things, like a name and the entity it represents.
- Binding is associate an attribute with an entity.
- Examples of attributes are name, type, value.
- ***Binding occurs at various times in the life of a program***
- The compiler performs a process called binding when an object is assigned to an object variable.

Binding time

- Binding time is the the moment when the binding is performed (compilation, execution, etc).
- The early binding (static binding) refers to compile time binding
- late binding (dynamic binding) refers to runtime binding.

The Concept of Binding

- The ***l*-value** of a variable is its **address**.
  The ***r*-value** of a variable is its **value**.
- A **binding** is an association, such as between an attribute and an entity,or between an operation and a symbol.

• A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.

• A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

Possible Binding Time

- Binding Time is the time at which a binding is created
  1. Language design time
  2. Language implementation time
  3. program writing time
  4. compile time
  5. link time
  6. load time
  7. *Runtime*

1. **Language design time** (bind operator symbols to operations. * to mul)
   - program structure, possible types , control flow constructs are chosen
2. **Language implementation time**
   - Coupling of I/O to OS, arithmetic overflow, stack size, type equality ,handling of run time exceptions
   - Ex)A data type such as **int** in C is bound to a **range** of possible values
3. **program writing time**
   - Programmers choose algorithms, data structures and names
4. **compile time**
   - bind a variable to a **particular data type** at compile time

5. link time

- Library of standard subroutines joined together by a linker.

6. load time

- Refers to the point at which the **OS loads the program into memory** so that it can run. virtual address are chosen at link time and physical addresses change at run time.
- bind a variable to a **memory cell** (ex. C **static** variables)

7.Runtime

- refers to the entire span from the beginning to the end of execution..virtual functions, values to variables, many more.
- bind a **nonstatic** local variable to a memory cell

Binding Time Examples

Language feature	*Binding time*
Syntax, e.g. `if (a>0) b:=a;` in C or `if a>0 then b:=a end if` in Ada	Language design
Keywords, e.g. `class` in C++ and Java	Language design
Reserved words, e.g. `main` in C and `writeln` in Pascal	Language design
Meaning of operators, e.g. + (add)	Language design
Primitive types, e.g. `float` and `struct` in C	Language design
Internal representation of literals, e.g. `3.1` and `"foo bar"`	Language implementation
The specific type of a variable in a C or Pascal declaration	Compile time
Storage allocation method for a variable	Language design, language implementation, and/or compile time
Linking calls to static library routines, e.g. `printf` in C	Linker
Merging multiple object codes into one executable	Linker
Loading executable in memory and adjusting absolute addresses	Loader (OS)
Nonstatic allocation of space for variable	Run time

Static & Dynamic Binding

- The terms static and dynamic are generally used to refer to things bound before run time and at run time, respectively
- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.
- Compiler-based language implementations tend to be more efficient than interpreter-based implementations because they make earlier decisions

Example: Static & Dynamic Binding

- Compiler is an example of static binding
- compiler analyzes the syntax and semantics of global variable declarations once, before the program ever runs.
- It decides on a layout for those variables in memory and generates efficient code to access them wherever they appear in the program.
- Interpreter is an example of dynamic binding
- A pure interpreter,by contrast,must analyze the declarations every time the program begins execution.
- interpreter may reanalyze the local declarations within a subroutine each time that subroutine is called

- When type of the object is determined at compiled time(by the compiler), it is known as static binding.

```
//static Example
class Dog{
 private void eat(){ System.out.println("dog is eating...");
}
public static void main(String args[])
{
  Dog d1=new Dog();
  d1.eat();
 } }
```

#dynamic example

```
class Animal{

 void eat(){System.out.println("animal is eating...");}

}

class Dog extends Animal
{   void eat()  { System.out.println("dog is eating...");}

  public static void main(String args[])
      {  Animal a=new Dog();

      a.eat();

       }  }
```

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type. o/p—Dog is eating

Dynamic Type Binding (JavaScript and PHP)

- Specified through an assignment statement
- Ex, JavaScript

  list = [2, 4.33, 6, 8];// single-dimensioned array

  list = 47;// scalar variable

- Advantage: **flexibility** (generic program units)
- Disadvantages:

  – **High cost** (dynamic type checking and interpretation)

  - Dynamic type bindings must be implemented using pure interpreter **not** compilers.

  -**Type error detection by the compiler is difficult** because **any** variable can be assigned a value of **any** type.

Effect of Binding Time

- **Early binding times** (before run time)are associated with **greater efficiency**
  - Syntactic and sematic checking can be done at compile time only once and run time overhead can be avoided
- **late binding times**(at run time) are associated with **greater flexibility.**
  - Interpreters allows programs to be extended at run time
  - Method binding in oops must be late to support dynamic binding.

Storage Bindings

- Storage Bindings & Lifetime
  - Allocation – getting a cell from some pool of available cells
  - Deallocation – putting a cell back into the pool

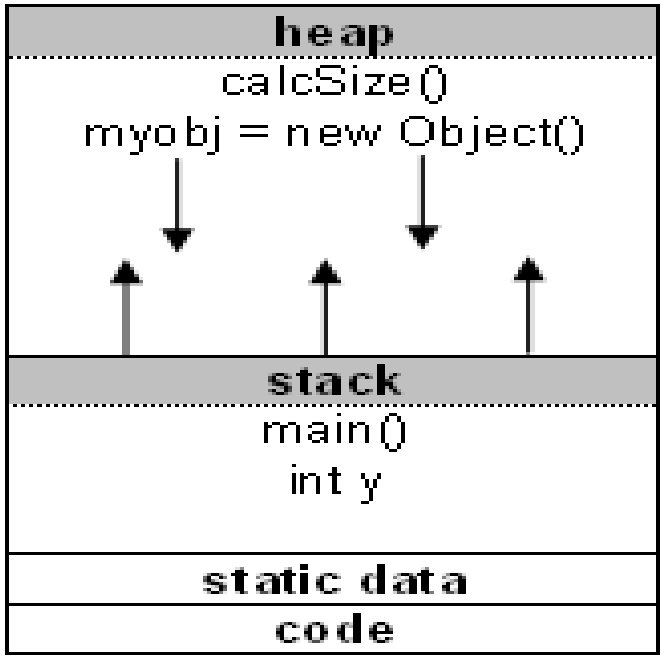
  - The lifetime of a variable is the time during which it is bound to a particular memory cell. To investigate storage bindings of variables, it is convenient to separate variables into four categories: (1) static, (2) stack-dynamic, (3) explicit heap-dynamic, and (4) implicit heap-dynamic.

Categories of Variables by Lifetimes (cont'd)

- (1).Static--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution, e.g., C and C++ `static` variables
  - Advantages: efficiency (direct addressing), history-sensitive subprogram support
  - Disadvantage: lack of flexibility (no recursion)

<Review of stack and heap>

- The stack is a place in the computer memory where all the variables that are declared and initialized *before* runtime are stored.
- The heap is the section of computer memory where all the variables created or initialized *at* runtime are stored.



```
int x; /* static stack storage */
void main() {
    int y; /* dynamic stack storage */
    char str; /* dynamic stack storage */
    str = malloc(50); /* allocates 50 bytes of
    dynamic heap storage */
    size = calcSize(10); /* dynamic heap storage */
    .
    .
    .
}
```

Categories of Variables by Lifetimes (cont'd)

- **(2).Stack-dynamic**--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)

  - As their name indicates, stack-variables are allocated from run-time stack.
  - In Java, C++, and C#, variables defined in methods are by default stack dynamic.
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Categories of Variables by Lifetimes (cont'd)

- **(3).Explicit heap-dynamic** -- 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

Categories of Variables by Lifetimes (cont'd)

- **(4).Implicit heap-dynamic**--Allocation and deallocation caused by assignment statements
  - all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection

Scope and Lifetime

Scope

- Scope - textual region of the program in which a binding is active
- most modern languages, the scope of a binding is determined statically
- Can be body of a module, class, subroutine, or structured control flow statement, sometimes called a block.
- In C family languages it would be delimited with {...} braces.

Scope

- Statically scoped language: the scope of bindings is determined at compile time.
- Dynamically scoped language: the scope of bindings is determined at run time.

Static Scoping

- determined by examination of the program text
- Scope rules of a program language define the scope of variables and subroutines
- C - Statically scoped

Static Scoping

- Basic Programming language:
  - all variables are global and visible everywhere
- Fortran77:
  - the scope of a local variable is limited to a subroutine;
  - the scope of a global variable is the whole program text unless it is hidden by a local variable declaration with the same variable name.

Static Scoping

- Algol60, Pascal, and Ada:

  these languages allow nested subroutines definitions and adopt the closest nested scope rule with slight variations in implementation.

Static Scoping

- C: one declares the variable static;
  - A saved (static, own) variable has a lifetime that encompasses the entire execution of the program.
  - Instead of a logically separate object for every invocation of the subroutine, the compiler creates a single object that retains its value from one invocation of the subroutine to the next.

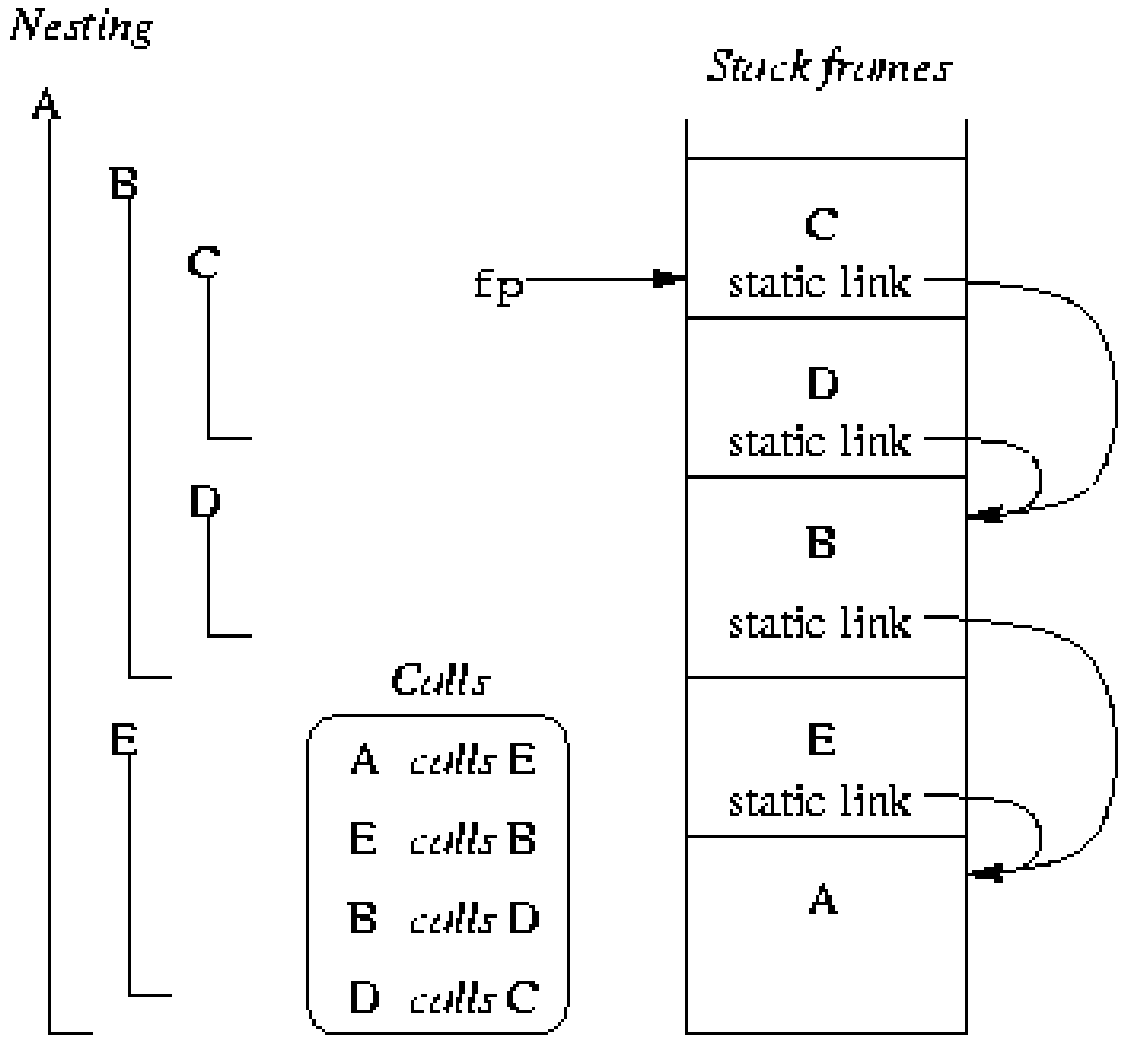
Static Scoping

- **Scope rules** are designed so that we can only refer to variables that are alive: the variable must have been stored in the frame of a subroutine
- If a variable is not in the local scope, we are sure there is a frame for the surrounding scope somewhere below on the stack:
  - The current subroutine can only be called when it was visible
  - The current subroutine is visible only when the surrounding scope is active

Static Scope Implementation with static links

- The simplest way in which to find the frames of surrounding scopes is to maintain a static link in each frame that points to the “parent frame”. Each frame on the stack contains a static link pointing to the frame of the static parent.

Static Scope Implementation with static links



Static Chains

- The static links forms a static chain, which is a linked list of static parent frames
- The compiler generates code to make these traversals over frames to reach non-local objects
- Subroutine A is at nesting level 1 and C at nesting level 3
- When C accesses an object of A, 2 static links are traversed to get to A's frame that contains that object

Out of Scope

```
procedure P1;
var X:real;
  procedure P2;
  var X:integer
  begin
    ... (* X of P1 is hidden *)
  end;
begin
  ...
end
```

- P2 is nested in P1
- P1 has a local variable X
- P2 has a local variable X that hides X in P1
- When P2 is called, no extra code is executed to inactivate the binding of X to P1

Dynamic Scoping

- determined by flow of execution of program
- APL, Snobol, Lisp etc
- in particular on the order in which subroutines are called

Static Vs Dynamic Scoping

```
1.  n : integer                  -- global declaration

2.  procedure first
3.      n := 1

4.  procedure second
5.      n : integer              -- local declaration
6.      first()

7.  n := 2
8.  if read_integer() > 0
9.      second()
10. else
11.     first()
12. write_integer(n)
```

Static versus dynamic scoping.

Program output depends on both scope rules and, in the case of dynamic scoping, a value read at run time.

Consider the program in Figure 3.9. If static scoping is in effect, this program prints a 1. If dynamic scoping is in effect, the output depends on the value read at line 8 at run time: if the input is positive, the program prints a 2; otherwise it prints a 1. Why the difference? At issue is whether the assignment to the variable n at line 3 refers to the global variable declared at line 1 or to the local variable declared at line 5. Static scope rules require that the reference resolve to the closest lexically enclosing declaration, namely the global n. Procedure first changes n to 1, and line 12 prints this value. Dynamic scope rules, on the other hand, require that we choose the most recent, active binding for n at run time.

Referencing Environments

Referencing Environment

- collection of all names that are visible in the statement
- In a static-scoped language, local variables plus all of the visible variables in all of the enclosing scopes
- In a dynamic-scoped language, local variables plus all visible variables in all active subprograms

Ex, Ada, **static-scoped language**

```
procedure Example is
   A, B : Integer;
   …
   procedure Sub1 is
      X, Y : Integer;
      begin      -- of Sub1
      …                          ← 1
      end        -- of Sub1
   procedure Sub2 is
      X : Integer;
      …
      procedure Sub3 is
         X : Integer;
         begin   -- of Sub3
         …                       ← 2
         end;    -- of Sub3
    begin  -- of Sub2
    …                            ← 3
    end;   { Sub2}
begin
   …                             ← 4
end;        {Example}
```

- The referencing environments of the indicated program points are as follows:

Point	***Referencing Environment***
1	X and Y of Sub1, A & 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

Ex, **dynamic-scoped language**

```
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 */
```

The referencing environments of the indicated program points are as follows:

Point	***Referencing Environment***
1	a and b of sub1, c of sub2, d of main
2	b and c of sub2, d of main
3	c and d of main

Binding of Referencing Environment

- Two types
  - Deep binding
  - Shallow binding

Deep Binding

- Binds the environment at the *time a procedure is passed as an argument*
- Example: f2(**f3**);
- Default in languages which uses static scoping
- It is the **early binding** (**static or lexical**) of the referencing environment of a subroutine
- The need for deep binding is sometimes referred to as ***the funarg problem in Lisp***

Deep Binding

- *Deep binding* **takes the environment of the parent function.**
- When **the procedure is passed as an argument, this referencing environment is passed as well**.
- When the procedure is eventually invoked (by calling it using the corresponding formal parameter), this **saved referencing environment is restored.**
- Deep Binding is to use the environment of the definition of the passed subprogram. Most natural for static-scoped languages.
- LISP function arguments and procedures in Algol and Pascal work this way.

Shallow Binding

- Binds the environment at the *time a procedure is actually called*
- Example:f3 ( ); function call
- Default in languages with dynamic scoping
- It is the **late binding (dynamic**) of referencing environment of a subroutine

Shallow Binding

- Shallow binding : **Takes the environment of the "final" calling function.**
- With *shallow binding* the nonlocal referencing environment of a procedure instance is the referencing environment in force at the time it (the procedure) is invoked.
- Original LISP works this way by default.
- ***No language with static (lexical) scope rules has shallow binding***
- **Shallow Binding is to use the environment of the call statement that CALLS the passed subprogram**.
- Most natural for dynamic-scoped languages.
    - Some languages with dynamic scope rules – only shallow binding (e.g., SNOBOL)
    - Others (e.g., early LISP) offer both, where default is shallow binding;

Deep binding (static)	Example program	Shallow binding (dynamic)
Point (1) **x of f1()□ 10**	**function f1()** **{** var x = 10;	Point (1) x of f1()□ 10
Point (2) x of f2()□ 6 (x of f1()hidden)	***function f2(fx)*** *{* *var x; x* *= 6;* ***fx();*** *};*	Point (2) **x of f2()□ 6** (x of f1()hidden)
Point (3) x of f1()□ 10 **Print x=10**	***function f3()*** *{* *print x;* *};*	Point (3) x of f2()□ 6 (x of f1()hidden) **Print x=6**
	f2(f3); **};**	