Programming Languages and Paradigms

Names, Scope, and Bindings

Topics

- Introduction
- Names
- Variables
- The Concept of Binding
- Type Checking
- Strong Typing
- Type Compatibility
- Scope and Lifetime
- Referencing Environments
- Named Constants

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory
  - Processor
- Variables characterized by attributes
  - Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

- Mnemonic character string used to represent sth.
- Most names are identifiers
- Symbols (like '+') can also be names
- Refer to variables, constants, operations, types, and so on

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

Variables Attributes

- Name - not all variables have them
- 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 created via pointers, reference variables, C and C++ unions
  - Aliases are harmful to readability (program readers must remember all of them)

Variables Attributes (continued)

- *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

Variables Attributes (continued)

- The l-value of a variable is its address
- The r-value of a variable is its value
  - Algol68: x = y + 1;
    - x : left-hand side value, *l-value,* its address
    - y : right-hand side value, *r-value,* its name
  - ML: x := !y + 1;
  - C/C++: int x, y, *p; x = *p;

Explicit dereferencing

The Concept of Binding

- A *binding* is an association between two things, such as
  - a name and the thing it names
  - formal parameters and actual parameters
  - an operation and a symbol

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

Possible Binding Times

- Language design time – operator symbols to operations
- Language implementation time – floating point type to a representation
- Compile time – a variable to a type in C or Java
- Link time – layout of whole program in (virtual) memory
- Load time – a C `static` variable to a memory cell
- Runtime – a non-static local variable to a (physical) memory cell, values to variables

Binding Example

C assignment statement

```
count = count + 5;
```

Examples of binding

Binding	Time
Type of `count`	Compile time
Set of possible values of `count`	Compiler design time
Meaning of operator symbol +	Compile time
Internal representation of `5`	Compiler design time
Value of `count`	Execution time

Static and Dynamic Binding

- 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
- Compiled languages tend to have early binding times for greater efficiency
- Interpreted languages tend to have later binding times for greater flexibility

Binding Example

- static int n = 2;
  - static binding, value of 2 to variable n
- int x;
  - static binding, integer data type to variable x
- x = 2;
  - dynamic binding, value of 2 to variable x

Type Binding

- How is a type specified?
- When does the binding take place?
- If static, the type may be specified by either an explicit or an implicit declaration

Explicit/Implicit Declaration

- An *explicit declaration* is a program statement used for declaring the types of variables
  - C: double sum = 0;

- An *implicit declaration* is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
  - Fortran: sum = 0
  - Perl: $i = 10

Dynamic Type Binding

- Dynamic Type Binding (JavaScript and PHP)
- Specified through an assignment statement
  e.g., JavaScript

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Advantage: flexibility (generic program units)
  - Disadvantages:
    - High cost (dynamic type checking and interpretation)
    - Type error detection by the compiler is difficult

Storage Binding & Lifetime

- Storage Binding
  - 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

Storage Binding & Lifetime

- Storage Allocation mechanisms
  - Static
  - Stack
  - Heap

Storage Binding & Lifetime

- Static – objects are given an absolute address that is retained throughout the program's execution

- Static allocation for
  - global variable
  - static or own variables
  - explicit constants (including strings, sets, etc)
  - scalars may be stored in the instructions

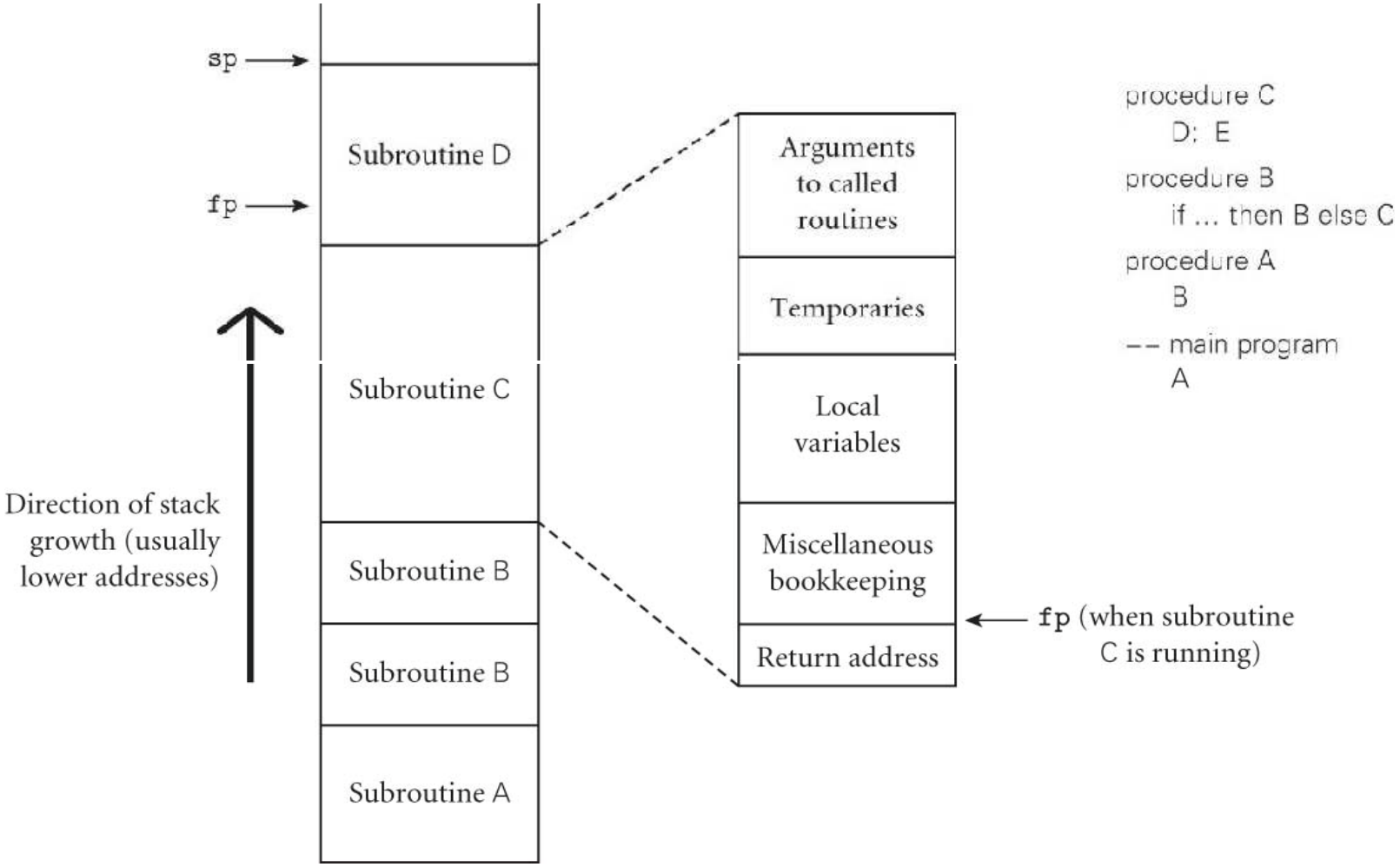
Storage Binding & Lifetime

- Stack – objects are allocated and deallocated in last-in, first-out order, usually in conjuction with subroutine calls and returns
- Stack allocation for
  - parameters
  - local variables
  - temporaries
- Why a stack?
  - allocate space for recursive routines
  - reuse space (in all programming languages)

Storage Binding & Lifetime

- Contents of a stack frame (figure in next slide)
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Stack for dynamic allocation



Storage Binding & Lifetime

- Heap – objects may be allocated and deallocated at arbitrary times.
- Heap allocation for
  - dynamically allocation pieces of linked data structure
  - Objects like strings, lists, and sets, whose size may changes during execution
- Require a more general (and expensive) storage management algorithm

Heap for dynamic allocation

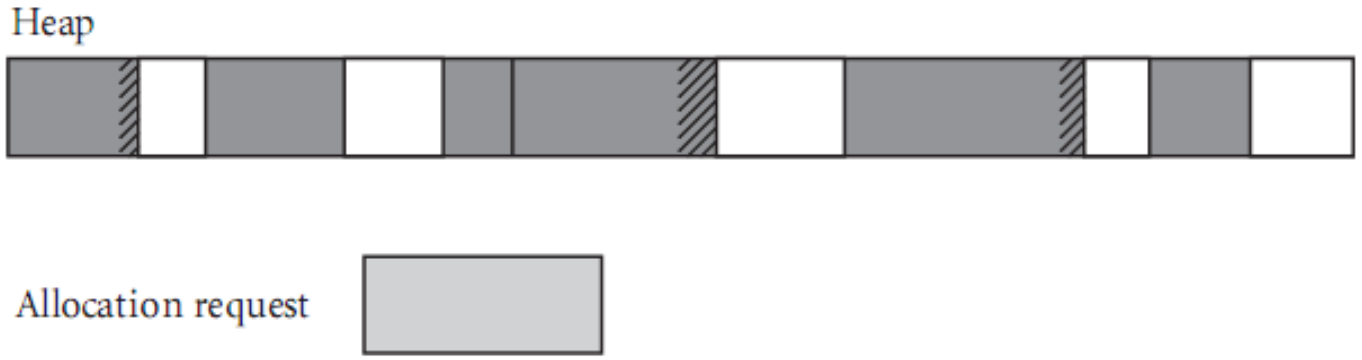


Figure 3.2 Fragmentation. The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments
- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types

Type Checking (continued)

- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type
  - This automatic conversion is called a coercion.
- A *type error* is the application of an operator to an operand of an inappropriate type

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if
  - type errors are always detected
  - types of all operands can be determined at compile time or at run time

Strong Typing

- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors
- Language examples:
  - FORTRAN 77 is not: parameters, `EQUIVALENCE`
  - Pascal is not: variant records
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked
  - Ada is, almost (`UNCHECKED CONVERSION` is loophole)

    (Java is similar)

Strong Typing (continued)

- Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)
- Although Java has just half the assignment coercions of C++, its strong typing is still far less effective than that of Ada

Type Checking Example

Pascal

```
Function  Even ( n : integer ) : boolean;
begin
    Even := (n mod 2) = 0
end;
```

Lisp

```
Function  Even ( n );
begin
    Even := n mod 2 = 0
end;
```

Name Type Compatibility

- *Name type compatibility* means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name
- Easy to implement but highly restrictive:
  - Subranges of integer types are not compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

Structure Type Compatibility

- *Structure type compatibility* means that two variables have compatible types if their types have identical structures
- More flexible, but harder to implement

Type Compatibility Example

```
TYPE arr_type = array[1..20] of
    integer;
VAR   x, y  :  array[1..20] of integer;
      a     :  arr_type;
      b     :  RECORD
                  c : integer;
                  d : arr_type;
               END;
```

A name equivalence b.d

```
CONST N = 2;
VAR     a : array[1..N] of char;
        b : array[1..N] of char;
```

A structure equivalence b

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *nonlocal variables* of a program unit are those that are visible but not declared there
- The scope rules of a language determine how references to names are associated with variables

Scope

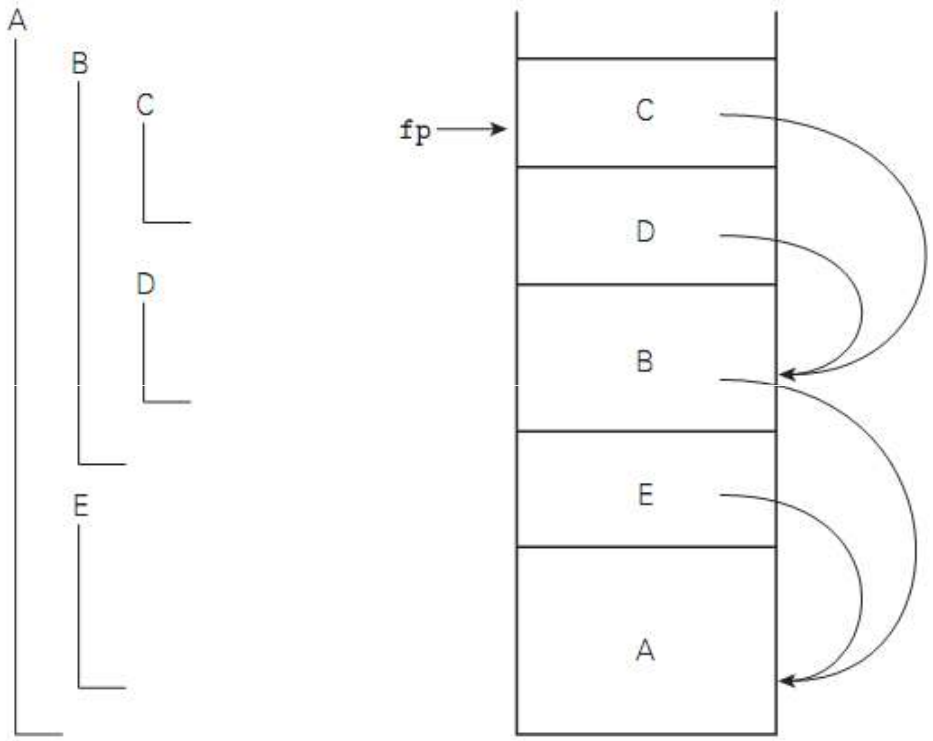


Figure 3.5 **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Scope Rules

- The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.
- With **dynamic scope rules**, bindings depend on the current state of program execution
  - They cannot always be resolved by examining the program because they are dependent on calling sequences
  - To resolve a reference, we use the most recent, active binding made at run time

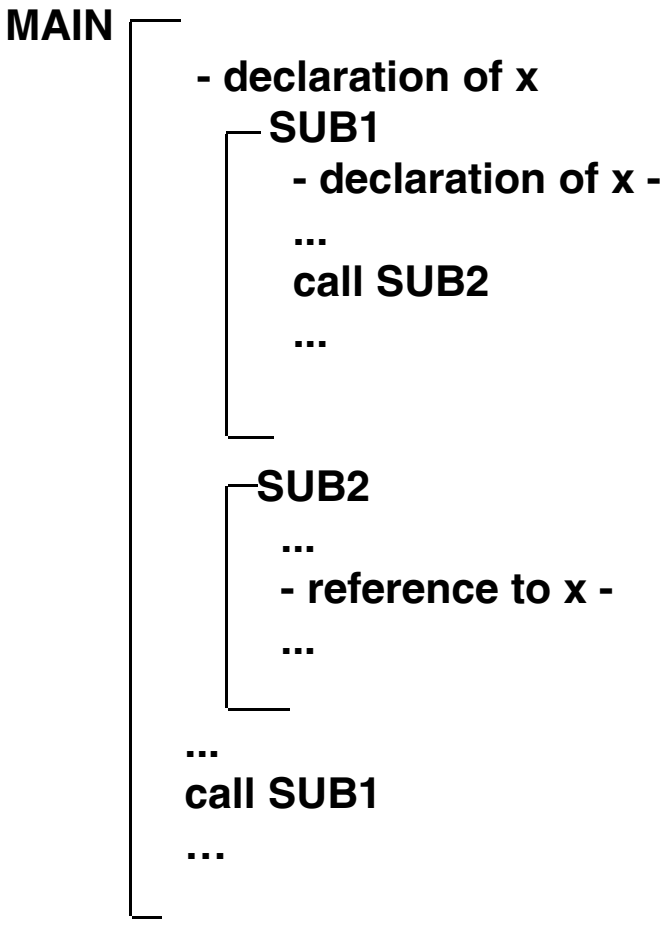
Scope Rule Example

```
program scopes (input, output );
var a : integer;
procedure first;
   begin a := 1; end;
procedure second;
   var a : integer;
   begin first; end;
begin
   a := 2; second; write(a);
end.
```

Scope Rule Example

- If static scope rules are in effect (as would be the case in Pascal), the program prints a `1`
- If dynamic scope rules are in effect, the program prints a `2`
- Why the difference? At issue is whether the assignment to the variable `a` in procedure *first* changes the variable `a` declared in the main program or the variable `a` declared in procedure *second*

Scope Example



MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Scope Example

- Static scoping
  - Reference to x is to MAIN's x
- Dynamic scoping
  - Reference to x is to SUB1's x
- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - Disadvantage: poor readability

Scope and Lifetime

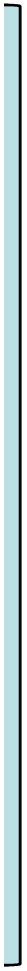
- Scope and lifetime are sometimes closely related, but are different concepts
- Consider a `static` variable in a C or C++ function

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, it is the local variables plus all of the visible variables in all of the enclosing scopes
- 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

Referencing Environments

```
Procedure Example is
    A, B : Integer;
    ….
    procedure  Sub1 is
        X, Y : Integer;
        begin
          ….
        end;
    procedure Sub2 is
        X: Integer;
        ….
        procedure Sub3 is
            X: Integer;
            begin
              ….
            end;
       begin
       ….
       end;
begin
…..
end;
```



Referencing Environments

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

Named Constants

- A *named constant* is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - FORTRAN 90: constant-valued expressions
  - Ada, C++, and Java: expressions of any kind

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called *initialization*
- Initialization is often done on the declaration statement, e.g., in Java

```
int 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