Chapter 4

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

Chapter 4

Case sensitivity

- *Disadvantage:* readability (names that look alike are different)
  - worse in Modula-2 because predefined names are mixed case (e.g. WriteCard)

- C, C++, Java, and Modula-2 names are case sensitive
- The names in other languages are not

Special words

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

- Disadvantage: poor readability

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

Chapter 4

A *variable* is an abstraction of a memory cell

Variables can be characterized as a sextuple of attributes:
name, address, value, type, lifetime, and scope

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 harmful to readability

Chapter 4

- ***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- **Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN**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

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**

Chapter 4

The *l-value* of a variable is its address
The *r-value* of a variable is its value

***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:

1. Language design time--e.g., bind operator symbols to operations

2. Language implementation time--e.g., bind fl. pt. type to a representation

3. Compile time--e.g., bind a variable to a type in C or Java

4. Load time--e.g., bind a FORTRAN 77 variable to a memory cell (or a C `static` variable)

5. Runtime--e.g., bind a nonstatic local variable to a memory cell

Chapter 4

***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 Bindings

1. **How is a type specified?**
2. **When does the binding take place?**

If static, type may be specified by either an explicit or an implicit declaration

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)

FORTRAN, PL/I, BASIC, and Perl provide implicit declarations

***Advantage:* writability**

***Disadvantage:* reliability (less trouble with Perl)**

Chapter 4

Dynamic Type Binding

- Specified through an assignment statement
  e.g. APL

```
LIST <- 2 4 6 8
LIST <- 17.3
```

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

Type Inferencing (ML, Miranda, and Haskell)

- Rather than by assignment statement, types are determined from the context of the reference

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

Chapter 4

Categories of variables by lifetimes

1. *Static*--bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.

e.g. all FORTRAN 77 variables, C `static` variables

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

***Disadvantage:* lack of flexibility (no recursion)**

2. Stack-dynamic--Storage bindings are created for variables when their declaration statements are elaborated.

- **If scalar, all attributes except address are statically bound**

e.g. local variables in Pascal and C subprograms

***Advantage:* allows recursion; conserves storage**

Disadvantages:

- **Overhead of allocation and deallocation**
- **Subprograms cannot be history sensitive**
- **Inefficient references (indirect addressing)**

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**

4. Implicit heap-dynamic--Allocation and deallocation caused by assignment statements
e.g. all variables in APL

Advantage: **flexibility**
Disadvantages:
- Inefficient, because all attributes are dynamic
- Loss of error detection

Chapter 4

Type Checking

- Generalize the concept of operands and operators to include subprograms and assignments

Def: *Type checking* is the activity of ensuring that the operands of an operator are of compatible types

Def: 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*.

Def: A *type error* is the application of an operator to an operand of an inappropriate type

- If all type bindings are static, nearly all type checking can be static

- If type bindings are dynamic, type checking must be dynamic

Def: A programming language is *strongly typed* if type errors are always detected

Chapter 4

Advantage of strong typing: **allows the detection of the misuses of variables that result in type errors**

Languages:

1. FORTRAN 77 is not: parameters, `EQUIVALENCE`

2. Pascal is not: variant records

3. Modula-2 is not: variant records, `WORD` **type**

4. C and C++ are not: parameter type checking can be avoided; unions are not type checked

5. Ada is, almost (`UNCHECKED CONVERSION` **is loophole) (Java is similar)**

Coercion rules strongly affect strong typing--they can weaken it considerably (C++ versus Ada)

Chapter 4

Type Compatibility

Def: ***Type compatibility by name*** **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)

Def: ***Type compatibility by structure*** **means that two variables have compatible types if their types have identical structures**

- More flexible, but harder to implement

Chapter 4

Consider the problem of two structured types:

- **Suppose they are circularly defined**
- **Are two record types compatible if they are structurally the same but use different field names?**
- **Are two array types compatible if they are the same except that the subscripts are different? (e.g. [1..10] and [-5..4])**
- **Are two enumeration types compatible if their components are spelled differently?**

- **With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)**

***Language examples*:**

Pascal: usually structure, but in some cases name is used (formal parameters)

C: structure, except for records

Ada: restricted form of name
- **Derived types allow types with the same structure to be different**
- **Anonymous types are all unique, even in:**

```
A, B : array (1..10) of INTEGER:
```

Chapter 4

Scope

Def: The *scope* of a variable is the range of statements over which it is visible

Def: 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

Static scope

- Based on program text

- 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*

Variables can be hidden from a unit by having a "closer" variable with the same name

- C++ and Ada allow access to these "hidden" variables

Blocks - a method of creating static scopes inside program units--from ALGOL 60

Examples:

C and C++:

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

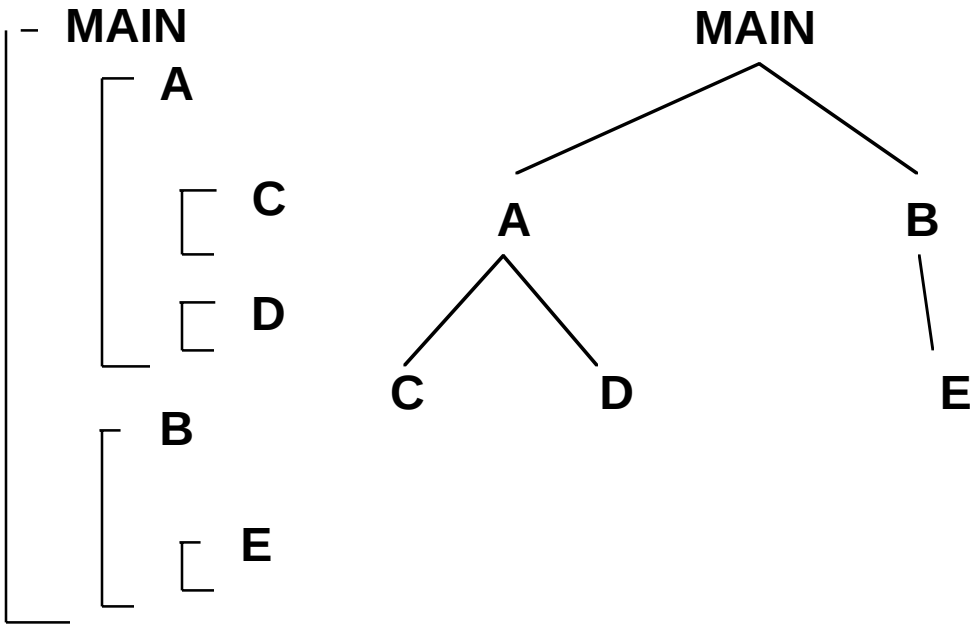
Ada:

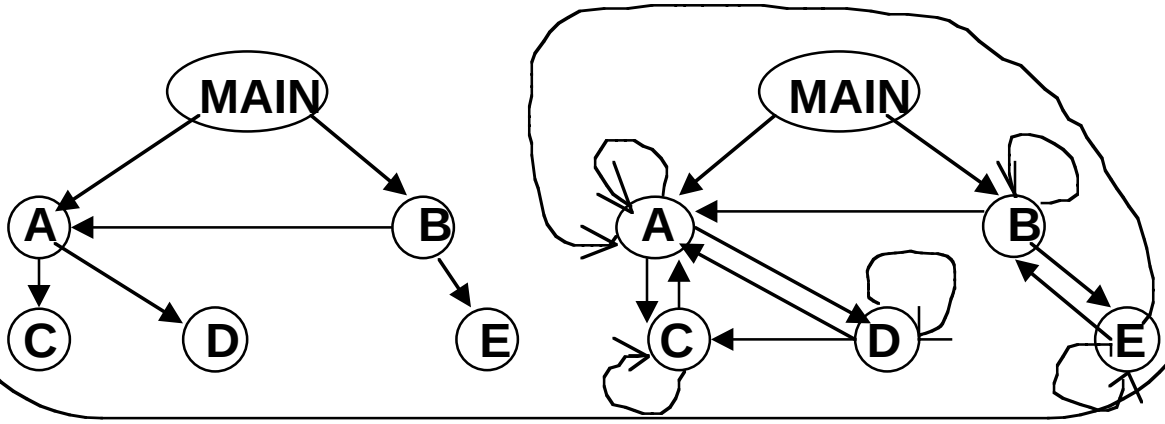
```
declare LCL : FLOAT;
   begin
            ...
   end
```

Chapter 4

Evaluation of Static Scoping

Consider the example:
Assume MAIN calls A and B
A calls C and D
B calls A and E





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**

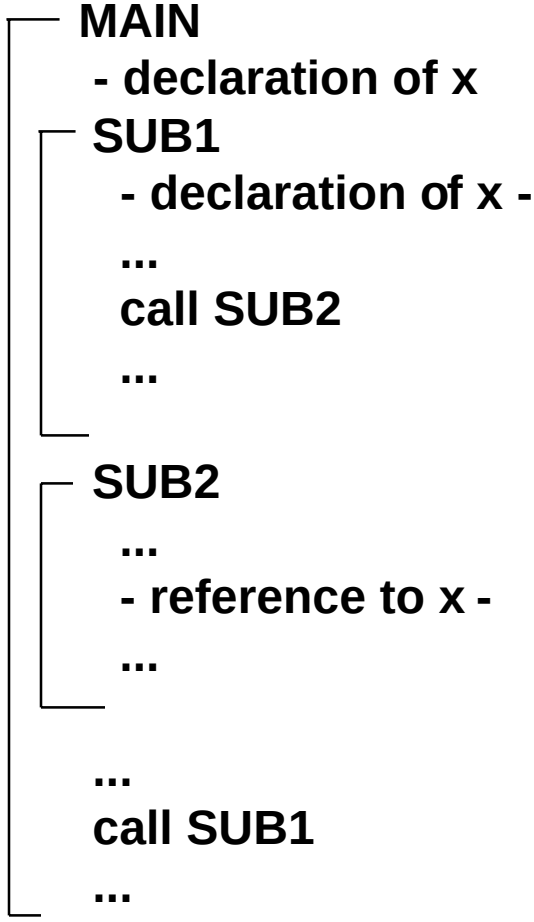
Dynamic Scope

- 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

Chapter 4

Example:



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

Chapter 4

Evaluation of Dynamic Scoping:
- ***Advantage:*** **convenience**
- ***Disadvantage:*** **poor readability**

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

Referencing Environments

Def: 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. 184)

- 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. 185)

Chapter 4

Def: A *named constant* is a variable that is bound to a value only when it is bound to storage

- *Advantages:* readability and modifiability

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**

Variable Initialization

Def: 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., Ada

```
SUM : FLOAT := 0.0;
```