Chapter 4 - Names, Bindings, Type Checking, and Scopes

Names (Identifiers)

- is a string of characters used to identify some entity in a program.
- can be associated with variables, labels, subprograms, formal parameters, etc.
- commonly acceptable name form is a string with a reasonably long length limit, with some connector character.

Primary design issues for names:

- Maximum length?
- Are connector characters allowed?
- Are names case sensitive?
- Are special words reserved words or keywords?

Name Forms

- **Length**
  - Earliest programming languages: single-character
  - 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
    - Why? For easy maintenance of symbol table.
- **Connectors**
  - Pascal, Modula-2, and FORTRAN 77 don't allow
  - Others do

- **Case sensitivity**

The difference between the cases of letters in names are recognized by the language.

  - disadv: readability (names that look alike are different)
  - diadv: sometimes also writability (in Modula-2 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
  - Prior to FORTRAN 90 only uppercase letters could be used
  - Many FORTRAN 77 implementations implicitly translates names to all uppercase letters

Special Words

- A **keyword** is a word that is special only in certain contexts
  - Disadvantage: poor readability
- 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 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.

Aliases. If two variable names can be used to access the same memory location, they are called **aliases**

  - Aliases are harmful to readability.

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- Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN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

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

4.4 The Concept of Binding

Binding is an association between an attribute and an entity or between an operation and a symbol.

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 (* ↔ multiplication operation)
2. Language implementation time – e.g., bind fl. pt. type to a representation (float ↔ range of possible values)
3. Compile time – e.g., bind a variable to a data type in C or Java (int count ↔ integer data type)
4. Link time – e.g., bind a call to a library subprogram to subprogram code.
5. Load time – e.g., bind a FORTRAN 77 variable to a memory cell (or a C static variable)
6. Runtime – e.g., bind a nonstatic local variable to a memory cell

Example: Bindings and Binding Times

```
int count;
. . .
count += 5;
```

- Set of possible types for **count**?
- Type of **count**?
- Set of possible values of **count**?
- Value of **count**?
- Set of possible meanings for the operator symbol +?
- Meaning of the operator +?
- Internal representation of literal 5?

Understanding of binding times for the attributes of program entities ⇒ understanding the semantics of a PL, e.g., . . .

4.4.1 Binding of Attributes to Variables

A binding is

- **static** if it occurs before run time and remains unchanged throughout program execution.
- **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 binding is static, type may be specified by either an explicit or an implicit declaration:

- An **explicit declaration** is a program statement used for declaring the types of variables
- An **implicit declaration** is a default mechanism for specifying types of variables (the first appearance of the variable in the program)
  - Perl and several of early languages (FORTRAN, PL/I, BASIC) provide implicit declarations
    - + writability
    - – reliability (less trouble with Perl): typographical and programming errors.

Dynamic Type Binding

A variable is bound to a type when it is assigned a value in an assignment statement. e.g. APL

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

+ flexibility (generic program units – capable of dealing with any type of data)

– type error detection by the compiler is difficult (incorrect types of rhs's of assignments are not detected as errors)
  – (This problem partially exists in PLs that use static binding such as C, C++, or FORTRAN.) How?

– high cost (dynamic type checking and interpretation)
  – variable-size descriptors associated with each variable

Interpretation is preferred because it is difficult to change dynamically the types of variables in machine code.

Type Inferencing in ML & Miranda

- Rather than by assignment statement, types are determined from the context of the reference
- **ML** (1990) is a recent language that supports both functional and imperative programming:

```
fun circumf ( r ) = 3.14159 * r * r;
fun times10 ( x ) = 10 * x;

fun square ( x ) = x * x;  // invalid!
fun square ( x ) : int = x * x;
fun square ( x : int) = x * x;
fun square ( x ) = (x : int) * x;
fun square ( x ) : x * (x : int);
```


Storage Bindings and 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

Categories of variables by lifetimes:

1. Static variables
2. Stack-dynamic variables
3. Explicit heap-dynamic variables
4. Implicit heap-dynamic variables

1 Static Variables

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

+ valuable applications, e.g., globally accessible variables
+ efficiency (direct addressing, no runtime overhead)
+ history-sensitive subprogram support
(retain values between separate executions)
– lack of flexibility (if only static vars ⇒ no recursion)

2 Stack-Dynamic Variables

Storage bindings are created for variables when their declaration statements are elaborated

- Elaboration of such a declaration refers to the run-time storage allocation and binding process indicated by the declaration.
- For scalar types, all attributes except address are statically bound, e.g. local variables in Pascal & C subprograms
- They are allocated from the run-time stack.

All variables are static in FORTRAN
(FORTRAN 77 support stack-dynamic variables)

+ allows recursion by supplying dynamic local storage
+ conserves storage - subprograms share same memory
– Overhead of allocation and deallocation
– Subprograms cannot be history sensitive
– Inefficient references (indirect addressing)

3 Explicit Heap-Dynamic Variables

Nameless objects that are allocated and deallocated from heap by explicit directives, specified by the programmer, which take effect during execution.

- Referenced only through pointers or references
- Created either by an operator (e.g., dynamic objects in C++ via **new** and **delete** & all objects in Java via **new**) or a system function (**malloc** and **free** in C)
- Bound to a type at compile time
- Bound to storage at run time

+ provides for dynamic storage management (e.g., linked lists, trees that grow and shrink during execution)
– unreliable and inefficient (references/(de)allocations)

4 Implicit Heap-Dynamic Variables

Allocation and deallocation caused by assignment statements, e.g. all variables in APL.

- In a sense they are just names that adapt to whatever use they are asked to serve.

+ flexibility (highly generic codes)
– inefficient, because all attributes are dynamic (including array subscript types and ranges)
– Loss of error detection

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.

- **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.

- **Static type checking:** If all type bindings are static, nearly all type checking can be static. (less costly & less flexible)
- **Dynamic type checking:** If type bindings are dynamic, type checking must be dynamic.

What if a memory cell can store values of different types at different times**?**

Strong Typing (1970s)

A simple/incomplete definition:

In a **strongly typed language**,

1. each name has a single type associated with it
2. and that type is known at compile time.

Although a variable's type is known, the storage allocation to which it is bound may store values of different types at different times. So we need a new definition:

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

- types of all operands can be determined either at compile time or run time.

+ we can detect all misuses of variables that result in type errors.

Type Strengths of Various Languages

1. FORTRAN 77 is not strongly typed: parameters, EQUIVALENCE.

2. Pascal is not: variant records without tag.

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)

6. ML and Miranda are strongly typed (In ML, types are all statically known either from declarations or from its type inference rules)

Coercion rules strongly affect strong typing--they can weaken it & error checking considerably (C++ vs. Ada)

Dynamic Type Binding

APL and SNOBOL4 languages.

The main advantage of dynamic type binding: programming: flexibility

Disadvantages:

1. efficiency
2. late error detection (costs more)

Type Compatibility Rules

… influence the data types and operations provided for objects of those types.

- name type compatibility
- structure type compatibility
- declaration equivalence

Type Compatibility

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:
  - Sub-ranges of integer types are not compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

Declaration equivalence: when a type is defined with the name of another type, the two are compatible, even though they are not name type compatible.

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

- More flexible, but harder to implement

Consider the problem of two structured types:

- Suppose they are circularly defined (i.e.,self-referential)
- 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?
- You cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Language Examples

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

C: structure eq., except for records (declarations are in separate files → structural type equivalence, otherwise declaration equivalence)

C++: name eq.

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:

Scope

The **scope** of a variable is the range of statements over which it is visible.
- A variable is **visible** in a statement if it can be referenced in that statement.

A variable is **local** in a program unit or block if it is declared there.

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 (determined statically, that is prior to execution)
- concept of "program units"

To connect a name reference to a variable, you (or the compiler) must find the associated 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:
- Pascal does not include nonprocedural blocks.
- C and C++ do not allow subprograms to be nested inside other subprogram definitions, but they have global variables. Local variables can hide these.
- C++ allow access to these "hidden" variables by using the scope operator (**::**).

Ex. on page 172.

Consider the presence of predefined names (keywords and reserved words.)

Blocks

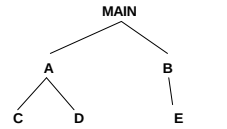
a method of creating static scopes inside program units – from ALGOL 60 - the first block-structured language!

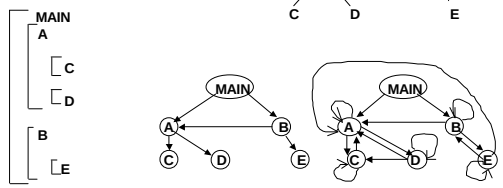
C and C++:
```
for (i=0;i<10;i++) {
    int index;
    ...
}
```

C and C++ allow any compound statement to have declarations, thus define a new scope.
- Local variables are all stack dynamic.
- C++ allows variable definitions to appear anywhere in functions.A variable's scope is from its definition statement to the end of the function.

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)
   - declaration of variables so far from their uses is harmful to readability.

Same problem for procedure access!

Overall: static scoping often encourages many globals.

Scope

(Fischer & Grodzinsky, 1993)

Scope in PL ⇔ Paragraph in an essay

- marked by a pair of matched opening and closing marks.

Correct Nesting:

Begin Scope A
  Begin Scope B
  End Scope B
End Scope A

Faulty Nesting:

Begin Scope A
  Begin Scope B
End Scope A
  End Scope B

Scope Example

```
INTEGER A, B, C(20), I
DATA A, B /31, 42/
READ* A, B, (C(I), I=1, 10)
DO 80 I=1, 10
    IF (C(I) .LT. 0) C(I+10)=0
    IF (C(I) .LT. 100) THEN
         C(I+10) = 2 * C(I)
    ELSE
         C(I+10) = C(I)/2
    ENDIF
80 CONTINUE
   END
```

Scopes are indicated in a variety of ways depending on the context:Dimension list, DATA values, implied DO, subscript list, DO loop, logical IF, block IF(true or false)

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.

- Search process: search declarations, first locally, then the declarations of the dynamic parent, or calling procedure, and so on until one is found for the given name.

Example

```
MAIN
   - declaration of x
   SUB1
    - declaration of x -
    ...
    call SUB2
    ...

   SUB2
    ...
    - reference to 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

Evaluation of Dynamic Scoping

- Advantage: convenience
- Disadvantages:
  - poor readability - calling sequence must be known.
  - inability to statically type check references to nonlocals
  - local variables of the subprogram are all visible to any other executing subprogram, regardless of its textual proximity.

Scope and lifetime are sometimes closely related, but are different concepts! Scope is textual, lifetime is temporal - 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, 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)

Named Constants

A named constant is a variable that is bound to a value only when it is bound to storage (i.e., its value can not be changed later on)

- 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

C: named literals with #define preprocessor command.

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., Ada

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

No initialization in Pascal.

- If the variable is statically bound to storage, binding and initialization occur before run time.
- If the storage binding is dynamic, initialization is also dynamic.