

Ch. 4: Names, Bindings, Type Checking, and Scopes

I. Introduction

- A. Procedural Languages are an abstraction of the underlying machine
- B. Two major components:
 - 1. Memory: stores programs & data. Abstraction of this: variables
 - 2. Processor: provides operations for modifying memory
- C. Variables characterized by a collection of properties (e.g. type, scope, lifetime)

II. Names

- A. Program entities given a name called an *identifier*
 - 1. Can refer to variable, constant, subprogram, parameter, type, or object
 - 2. An identifier is bound to a *referent*, which is the actual entity
 - 3. Identifier-to-referent correspondence is not one-to-one;
 - e.g. int x can be different values
 - Identifiers should be mnemonic
- B. Design issues: What is max. length? Can connector chars. be used? or keywords?
- C. Name forms

A name is a string of chars. used to identify some entity in a prog.

- 1. Length
 - Originally only single char. names
 - FORTRAN: 6 char. names,
 - FORTRAN 90 & C: up to 31 chars.
 - Ada, C++: no limit (implementations vary)
- 2. Case-sensitivity
 - C, C++, Modula-2 are case-sensitive
 - E.g. int lf// Valid in C
 - This violates design principle: similar appearance, similar function Words

D. Special Words

- 1. Reserved words: may not be redefined by programmer, special use
- 2. Keywords: special only in certain contexts
 - E.g. FORTRAN: REAL declares type or can be variable
 - REAL SALARY
 - INT REAL
 - REAL = 3.14159
 - REAL INT
- 3. Predefined Names: In between reserved words & user-defined names
 - Have predefined meanings but can be redefined by user
 - a. E.g WRITE & WRITELN in Pascal
 - b. C & C++: predefined names in libraries

E. Declarations

1. Declaration is non-executable statement introducing an identifier into some prog. scope
 - Statically typed lang.: also give *type* of referent
 - C++ allows declarations at any point
2. Declaration vs. *definition*
 - Definition instantiates the identifier, binding name w/the entity.
 - e.g. A person's name, as opposed to an actual person with that name
3. Forward declarations: Using an identifier that is defined later in program
 - Used with mutually referential subprograms, types pointing to each other.
 - E.g. 1: C function declaration before main, when body of function is after main
 - E.g. 2: Function is external [see overhead]
4. Symbol table: stores attributes & bindings of program identifiers
 - a. Stores location: stack frame offset for local variable
 - b. Symbol table exists only during compilation for compiled lang.
 - Must exist during execution for interpreted languages

III. Variables: Can be characterized as a sextuple of attributes: (name, address, value, type, lifetime, scope)

A. Name: i.e. identifier

B. Address:

l-value: l.h.s. of assignment: the address of storage location.

- Used for passing parms. by reference

- different possibilities:

1. Name associated w/different addresses at different places in the prog.
 - E.g. In C, two fcns. f1 & f2 both use local var. i
2. Same name associated w/different addresses at different times in prog.
 - E.g. same var., but recursion determines which stack frame to use
3. Same name may be dynamically bound to a different address.
 - E.g. In C, a fcn. local var. may be dynamically bound to different memory each time it is called
4. Aliases: Multiple identifiers reference same address
 - a. pointers, variant structures, unions
 - b. Justification for this no longer exists: dynamic allocation replaces need to use same memory for different purposes

C. Type: Determines the

1. Range of allowable values
2. Set of operations for that type
3. How storage will be interpreted

D. Value: abstraction of the actual physical implementation details

r-value: r.h.s. of assignment: results in a value. Used for passing parms. by value

IV. Binding

This is an association between an attribute and an entity, or between an operation and a symbol

A. Data objects are information that must be represented by the application

We must differentiate between:

1. Program Object: Program's logical named entities; names are *identifiers*, entities are *types*
2. Storage Object: How the program object is actually stored in memory; An *instantiation* of the logical entity

B. Binding time possibilities:

1. Language Design time. E.g. * bound to mult.
2. Language Implementation time. E.g. int bound to a range of values
3. Compile time. E.g. a variable is bound to a type when compiled
4. Link time. E.g. a call to a pre-compiled system fcn., such as printf
5. Load time. E.g. the actual memory address assigned to a variable in relocatable code
6. Run time. E.g. Allocation of memory for local function variables on the stack

C. E.g. C assignment statement:

```
int count;
```

```
...
```

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

1. Set of possible types for count: bound at lang. design time
2. Type of count: bound at compile time
3. Set of possible values for count: bound at compiler design time
4. Value of count: bound at execution time of this statement
5. Set of possible meanings for "+": bound at lang. definition time
6. Meaning of this "+" symbol: bound at compile time
7. Internal representation of the literal 5: bound at compiler design time

D. Binding of attributes to variables

May be *static* (occurs before run time & remains unchanged) or *dynamic* (occurs during run-time or may change during execution)

E. Type bindings: before a var. is used it must be bound to some type at some point in time

1. Variable declarations: May be explicit (C, Pascal), or implicit (FORTRAN, BASIC, ML)
2. Dynamic Type Binding: Binding determined when executed
 - a. Advantage: Programming flexibility. Can write a general operation independent of underlying type (C++)
 - b. Disadvantage:
 - (1). Error detection capability is diminished. Type mismatches not flagged
 - (2). Cost: expensive to implement, usually done in interpreters
3. Type Inference. E.g. ML
 - a. fun circumf (r) = 3.14 * r * r;//infers type of result
 - b. fun circumf (x) = x * x;//not allowed, type info. missing
 - c. fun circumf (x) : int = x * x;// fixes the above problem

F. Storage Bindings and Lifetime

When is storage allocated? When is it deallocated?

1. Static Variables: bound to memory cells before program execution begins
 - a. May be "historically sensitive" (E.g. C static vars. in a fcn.)
 - b. Advantage: efficiency. Disadvantage: loss of flexibility, recursion not supported
2. Automatic (Stack-Dynamic) Variables: bound to memory cells when execution reaches that part of the program (i.e. it is elaborated)
 - a. E.g. Pascal: variables defined in a subprogram, nested lifetimes [overhead]
 - b. This permits recursion
3. Dynamic (Explicit Heap-Dynamic) Variables: allows for programmer-controlled run-time allocation of variable amounts of memory (e.g. for trees, lists, graphs; using **malloc** and **new**) from the *heap* (a.k.a. *free store*)
 - a. Allocation Concerns: Memory management system must have:
 - (1). compaction to handle fragmentation
 - (2). free space allocation algorithm: best fit, next fit, worst fit, buddy system
 - b. Deallocation of dynamic storage
 - (1). **dispose** and **free**
 - (2). *memory leak*: programmer forgets to release storage
 - (3). *dangling pointer*: referring to previously released storage
 - (4). *automatic storage reclamation*: run-time recovery of unreferenceable storage.
May use *reference counting*
 - (5). Recovery of cyclically linked unreferenceable storage known as *garbage collection*.
 - c. Advantage: allows for great flexibility, avoids use of unions & variant records.
 - d. Disadvantage: Loss of simplicity; Overhead
4. Lifetime of an Object
 - a. Variable definitions or dynamic allocation (e.g. malloc, new) instantiate the data type (declaration) and allocate storage (definition) [*dec.* before *def.*]
 - b. Lifetime != scope: when in subprogram, variable may not be accessible, but still exists
5. Initialization and Assignment
 - *Initialization* stores a meaningful value immediately after creation (none in Pascal)
 - *assignment* replaces current value with a new value

V. Type Checking

- To include subprograms in this discussion, consider them to be operators whose operands are their parameters.

- A. Type Checking: verifying that operands are of compatible types, either explicitly or by *coercion*
- B. Allowing the same memory to have different types complicates this (e.g. C unions)

VI. Strong Typing

A. A Programming Language is *strongly typed* if type errors are always detected.

This implies that the types of all operands can be determined either at compile or run time.

B. Examples:

- 1. Pascal is nearly strongly typed, though it allows variant records
- 2. Ada is nearly strongly, though allows explicit UNCHECKED_CONVERSION
- 3. C is not strongly typed, since function parameters are not type checked
- 4. ML & Miranda are strongly typed

C. Coercion weakens the value of strong typing

- 1. Mixing int & float in C *promotes* the int to float. This can be an undetected error
- 2. Ada mixed mode arithmetic must have an explicit *cast* to work (there is less coercion)

VII. Type Compatibility

A. Types of subprograms are their *argument signature*

B. Two types: Name Compatibility and Structure Compatibility

C. Name Compatibility: two variables have compatible types only if they have the same type name

1. E.g. in Pascal, assuming strict name compatibility:

```
TYPE indexType = 1 .. 100; { a subrange type}
VAR
```

```
    count: INTEGER;
```

```
    index: indexType;
```

the variables *count* and *index* are not compatible

D. Structure Compatibility: two variables have compatible types only if they have the same structures.

1. Determining compatibility of two structures may be difficult, for instance:

- a. Same structure, but different field names
- b. Same array element type & size, but different subscripts (0..9 vs. 1..10)

2. Structure compatibility prevents differentiation between types of same structure.

E.g.

```
TYPE length = REAL;
```

```
    area = REAL;
```

These could be mixed (are compatible), even though they mean different things.

3. Structural vs. Name compatibility may not always be clear. Example from Pascal:

```
TYPE
```

```
    type1 = ARRAY [1..10] OF INTEGER;
```

```
    type2 = ARRAY [1..10] OF INTEGER;
```

```
    type3 = type2;
```

Here type1 & type2 in Pascal are not compatible (not structure compatibility), although type2 & type3 are compatible (not strict name compatibility either).

E. Defining a type with the name of another type is called *declaration equivalence*.
(e.g. given above of type3 = type2)

F. Ada uses name type compatibility, but provides *subtypes* and *derived types*.

1. E.g. of derived type:

```
TYPE celsius IS NEW FLOAT;  
TYPE fahrenheit IS NEW FLOAT;
```

Here celsius and fahrenheit are derived from FLOAT, and so *are not* compatible

2. E.g. of subtype:

```
SUBTYPE Small_Type IS INTEGER RANGE 0..99;
```

Here Small_Type *is* compatible with other INTEGERS.

G. C uses structural equivalence for all types except structs, for which it uses declaration equivalence

H. Anonymous types: a type is defined, but not named. E.g. declaring array A in Ada:

```
A: ARRAY (1..10) OF INTEGER;
```

VIII. Scope

A. Definitions:

1. *Scope* is the range of statements in which a variable is visible

2. A variable is *visible* if it can be referenced. References associate variables with their declarations and thus their attributes

3. A variable is *local* to a program unit if it is declared there. A variable is *nonlocal* if it is visible within a program unit but not declared there.

B. Blocks - demarcated sections of code. E.g.: Subprograms in Pascal, { } in C

1. Variables may be defined locally within a block (1st done in ALGOL 60) E.g.

```
if (list[i] < list[j]) {  
    int temp = list[i];  
    list[i] = list[j];  
    list[j] = temp;  
}
```

temp is an example of a stack variable.

2. C++ allows variables to be defined anywhere in a function, with scope from the point of declaration to the end of the function.

C. Static Scope [See Pascal Example]

1. Process: First look locally (in local_proc). If not there, look in the static parent (the procedure that declared local_proc). Continue following static parents until either it is found or the top level is reached.

2. The collection of static parents are static ancestors.

a. Accessing nonlocal variables involves following *static links* into stack frames (discussed later in course).

b. Identifier resolution can see *outward*, but not *into* a scope

3. Problem: predefined names: keywords must be searched before local scope declarations

4. Hiding variables:

sub2 can't access the y of sub1

sub2 can't access the x of main

5. A *qualified name* allows specifying both scope and name:

a. ADA: scope can be specified, as in: scope.name

b. C++: scope can be specified, as in scope::name

D. Evaluation of Static Scope

1. Scope of [Fig. 4.1] can be represented as a tree [Fig. 4.2]
2. Actual possible calls are more accurately represented by a graph [Fig. 4.3]. Two problems:
 - a. Calls to a procedure that should not have been callable will not generate an error
 - b. Putting all variables in the main program makes data visible within scopes where it should not be.
3. Consider these scenarios in our Pascal example, where sub3 needs access to variable y of sub1:
 - a. Move sub3 into the scope of sub1
 - b. Make the variables global
4. To get this to work, we end up putting everything at the outer-most level (global) of scope, which is not what we want

E. Dynamic Scope.

1. When the search of local declarations fails, dynamic scoping searches the declarations of the dynamic parent (the calling procedure).
2. Again consider the Pascal example, this time assuming dynamic scoping is used: If main calls sub1 calls sub2 and sub2 references y, then once sub2 is checked (not found in local scope), the next place to check is *not* the static parent (main), but the dynamic parent (sub1). Only if it is not found there is main checked.

F. Evaluation of Dynamic Scoping

1. Dynamic scoping implies that attributes of a variable will not always be the same. (Consider trying to determine the semantics...)
2. The variables in the calling procedure will always be visible to the called procedure. This can be bad (poor encapsulation), but can also be used as a communication mechanism.

IX. Scope and Lifetime

- A. It seems that scope and lifetime may be the same, e.g. a variable defined within a block may only exist for the duration of that block
- B. Difference between the two:
 1. Static scope is a textual, or spatial concept
 2. Lifetime is a temporal concept
- C. Counter Examples to the idea that scope and lifetime are the same:
 1. A C function variable declared as **static** has permanence (allocated statically, not automatically)
 2. Subprogram calls;

```
void printhead ()  
{.....}
```

```
void compute () {  
    int sum;  
    ...  
    printhead ();  
} // end of compute
```

Here "sum" has lifetime during the call to printhead, although it is not within the scope of printhead.

X. Referencing Environments

- A. The referencing environment of a statement in a static-scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible. [See example]
- B. The referencing environment of a statement in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms that are currently *active*. (An *active* subprogram is one that has begun but has not yet terminated.) [See example]

XI. Named Constants

- A. A named constant is a variable whose value is bound at the time it is bound to storage. Its value cannot be changed by an assignment or by an input statement.
- B. Advantages: readability, use in multiple declarations (for loop counters throughout program, array bounds)

C. Examples:

1. Ada:

```
MAX: constant integer := 2 * WIDTH + 1;
```

Note the use of an expression on r.h.s. of assignment

2. C:

```
#define LISTLEN 100
```

XII. Variable Initialization

- A. Initialization is the binding of a value to a variable at the time it is bound to storage
- B. This is done before run time for static variables, during run time for dynamic variables
- C. Examples:

a. In C:

```
int counter = 0;    // Initialize value when storage is declared
```

b. ALGOL 68:

```
int first := 10;    { a variable }
```

```
int second = 10;   { a constant }
```

Note the poor design of two similar-looking constructs have different meanings.

c. Ada

```
LIMIT: constant INTEGER: = OLD_LIMIT + 1;
```