Chapter 5

Names, Bindings, Type Checking, and Scopes

"A variable can be characterized by a collection of properties, or attributes, the most important of which is type. The design of the data types of a language requires that a variety of issues be considered : scope, lifetime of variables, type checking, and initialization".

5.1 Introduction

- *Imperative Languages*
  - *are abstraction of the underlying von Neumann computer architecture*
    - ⇔ *memory : stores both instruction and data*
    - ⇔ *CPU : provides the operations for modifying the contents of the memory*
  - *Variable* in an imperative language
    - ⇔ an abstraction of memory cell
    - ⇔ can be characterized by a collection of properties (or *attributes*)
      - ⇒ data types, scope, life time, type checking, initialization,…
- How well the data types match the real-world problem space ?

5.2 Names

- a name is a string of characters used to identify some entities (variables, labels, subprograms, and formal parameters) in a program

- Design Issues
  - What is the maximum length of a name ?
  - Can connector (underscore) characters be used in names ?
  - Are names case sensitive ?
  - Are the special words, reserved words or keywords ?
- Name Forms
  - *Name length*
    - ⇔ Earliest programming languages : used single-character names (name for unknown)
    - ⇔ FORTRAN I : allowing upto 6 characters
    - ⇔ COBOL : allowing upto 30 characters
    - ⇔ C#, Ada, and Java: no limit, and all are significant
  - *Case sensitive*
    - ⇔ C (C++) and Modular recognize the difference between the cases of letters in names
    - ⇔ *Is it a good feature ?*
- Special Words : an aid to readability; used to delimit or separate statement clauses
  - *Keyword* : a word of a programming language that is special only in certain contexts
    - ⇔ In FORTRAN
      - ⇒ REAL APPLE
      - ⇒ REAL = 3.4
  - *Reserved word* : a special word of a programming language that can not used as a name *(Reserved words are better than keywords for readability) (COBOL has 300 reserved words!)*

5.3 Variables

- an abstraction of a computer memory cell, or a collection of cells
- can be characterized as a sextuple of attributes (name, address, value, type, lifetime, scope)

- *Name*
  - Variable name is the most common names in programs
  - often referred to as identifiers
- *Address*
  - The address of a variable is the memory address with which it associated
  - In many languages, it is possible for the same name to be associated with different addresses at different places in the program (*local variable*)
  - When more than one variable name can be used to access a single memory location, the names are called *aliases*
    - ⇔ In FORTRAN, through *EQUIVALENCE* statement
    - ⇔ In Pascal, through *variant* record structures
    - ⇔ through *subprogram parameters*
    - ⇔ through usage of *pointers*
- *Type*
  - The type of variable determines the *range of values* the variable can have and *the set of operations* that are defined for values of the type
    - ⇔ 예) Integer in FORTRAN : -32,768~32,767, arithmetic operations
- *Value*
  - The value of variable is the *contents of the abstract memory cell* associated with the variable
    - ⇔ *l-value* : the address of a variable
    - ⇔ *r-value* : the value of variable

a = a + 1

l-value *r-value*

5.4 The Concept of Binding

- a binding is an association, such as between an *attribute and an entity* or between *an operation and a symbol*
- **Binding Time** : *the time at which a binding takes place*
  - at language design time
    - ⇔ '*' is usually bound to the multiplication operator
  - at language implementation time
    - ⇔ a data type (such as Integer) is bound to a range of possible values
  - at compile time
    - ⇔ a variable in a Pascal program is bound to a particular data type
  - at link time
    - ⇔ a call to a library subprogram is bound to the subprogram code
  - at load time
    - ⇔ a variable may be bound to a storage cell when the program is loaded into memory
  - at run time
    - ⇔ a variable in a procedure may be bound to a storage cell when the procedure is actually called

예제

```
int count ;
….
count = count * 5;
sub1(count);
….
sub1(int aa) { …. }
```


(1) Binding of Attributes to Variables

- a binding is
  - *static* if it occurs *before run time* and *remain unchanged* throughout program execution
  - *dynamic* if it occurs during *run time* or *can change* in the course of program execution

(2) Type Binding

- *Static Type Binding (Variable Declaration)* :
  - *How the type is specified ?*
    - ⇔ an *explicit* declaration is a statement in a program that lists variable names and declare them to be of particular type
      - ⇒ most programming languages designed since the mid-1960 requires explicit declaration of all variables
    - ⇔ an *implicit* declaration is a means of associating variables with types through default convention
      - ⇒ FORTRAN (`I,J,K,L,M,N`), BASIC, PL/I
- *Dynamic Type Binding* (JavaScript, Python, Ruby, PHP, and C# (limited))
  - the variable is bound to a type when it is assigned a value in an assignment statement.
  - usually implemented using interpreters because it is difficult to dynamically change the type of variables in machine code
  - Advantages
    - ⇔ Flexibility (Generic program)
  - Disadvantages
    - ⇔ Hard to detect errors at compile time
    - ⇔ Run time cost for type checking, and space cost for tag

```
count = 1;
…
count = 3.0;
…
count = [1,2,3] ;
```

JavaScript

- *Type Inference*
  - *in ML,*

```
fun circum(r) = 3.14*r*r;
```

(3) Storage Binding and Lifetime

- the *lifetime* of a program variable is the time during which the variable is bound to a specific memory location

- ***Static Variables***
  - static variables are those that are bound to memory cells before execution begins and remain bound to those same memory cells until execution terminated
    - ⇔ global variables, history sensitive variable
  - advantages : efficiency
  - disadvantages : reduced flexibility (can not support recursion)

- ***Stack Dynamic Variable***
  - stack dynamic variables are those whose storage bindings are created when their declaration statement are elaborated, but whose types are statically bound
    - ⇔ the local variables declared in a procedure
  - advantages:
    - ⇔ allowing recursion
    - ⇔ sharing of the same memory cells between different procedures
  - disadvantages
    - ⇔ run time overhead to allocate and deallocate the memory cells for local variables

- ***Explicit Heap Dynamic Variables (by programmer)***
  - Explicit dynamic variables are nameless objects whose storage is allocated and deallocated by explicit run-time instructions specified by the programmers (referenced via pointer variables)
  - It is bound to a type at compile time, but is bound to storage at the time it is created
  - Example

```
type intnode = ^integer ;
var anode : intnode ;
...
new(anode) ;
...
dispose(anode) ;
....
```

Pascal : procedure
Ada : operator
C : function (`malloc()`)
C++ : `new` and `delete`

  - advantages
    - ⇔ used for dynamic structures
  - disadvantages
    - ⇔ difficulty of using them correctly and the cost of references, allocations, and deallocations ➔ *inefficient and unreliable*

- ***Implicit Heap Dynamic Variables (by system)*** *(JavaScript, and PHP)*
  - Implicit dynamic variables are bound to storage only when they are assigned values
  - Example: (in APL)

```
LIST = 10.2 5.1 0.0
LIST = 47
```

memory allocation for list
memory allocation for integer

  - advantages
    - ⇔ high flexibility
  - disadvantages
    - ⇔ run time overhead of maintaining all the dynamic attributes, loss of error detection

5.5 Type Checking

- *Type checking* is the activity of ensuring that the operands of operator are of compatible types
- A compatible type is one that is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code to a legal type (coercion)
- If all binding of variables to types are static in a language, then type checking can nearly always be done statically (static type checking)
- Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution (Pascal variant records)

5.6 Strong Typing

- Strongly typed language
  - each name in a program in the language has a single type associated with it, and that type is known at compile time (all types are statically bound)
  - type errors are always detected
- Example
  - FORTRAN 77 : not strongly typed because the relationship between actual and formal parameters is not type checked
  - Pascal : nearly strongly typed, but fails in its design of variant record
  - C, ANSI C, C++ : not strongly typed languages because all allow function for which parameters are not type checked
  - Ada : nearly strongly typed language

5.7 Type Equivalence

- Two variables are type compatible *if either one can have its value assigned to the other*
  - Simple and rigid for predefined scalar type
  - In the case of structured type (array, record,..), the rules are more complex
- Two types are equivalence *if an operand of one type in an expression is substitute for one of the other, without coercion*
  - A restricted form of type compatibility (type compatibility without coercion)
  - There are two approaches to defining type equivalence

    ⇔ Name Type Equivalence : two variables have equivalence types if they are defined either the same declaration or in declarations that use the same type name (Pascal)

    ⇒ Easy to implement, but highly restrictive

```
type indextype is 1..100
count : Integer;
index : indextype;

count := index; /* ➔ Error !! */
```

    ⇔ Structure Type Equivalence : two variables are compatible type if their types have identical structures (Ada)

    ⇒ More flexible, but difficult to implement

```
subtype Small_type is Integer range 0..99;

/* Small_type is equivalent to the type Integer */
```

5.8 Scope

– The *scope* of a program variable is the range of statement in which the variable is visible
– A variable is *visible* in a statement if it can be referenced in that statement
– The scope rules of a language determine how references to names are associated with variables

⇔ static scoping, dynamic scoping

(1) Static Scoping

– the scope of variable can be statically determined (ALGOL60, ...)
– Suppose a reference is made to a variable X in procedure A. The correct declaration is found for the variable there, the search continues in the declaration of the procedure that declared procedure A (static parent) until a declaration for X is found

```
procedure big ;
  var x : integer ;
  procedure sub1 ;
    begin
      ... x ....
    end ;
  procedure sub2;
    var x : integer ;
      begin
        sub1 ;
      end ;
  begin
    sub2 ;
  end ;
```

big -> sub2 -> sub1

nested procedure definition (ex, Pascal)

- The local variables of a program unit are those that are declared in that unit
- The nonlocal variables of a program unit are those that are visible in the unit but not declared there
- global variables are a special category of nonlocal variables

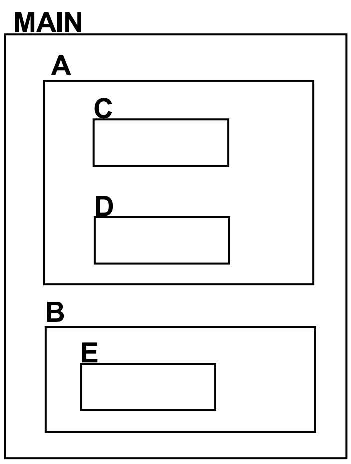
- Some languages allow new static scopes (called blocks) to be defined in the midst of executable code
  – allows a section of code to have its own (semidynamic) local variables
  – compound statement in C

```
if (a < b) { int temp ;
             temp = a ; a = b ; b = temp; }
```

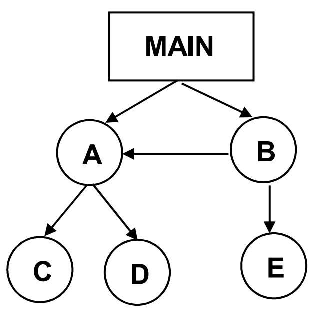
- Ada
- JavaScript
- Fortran 2003+
- F#
- Python)

nested subprogram

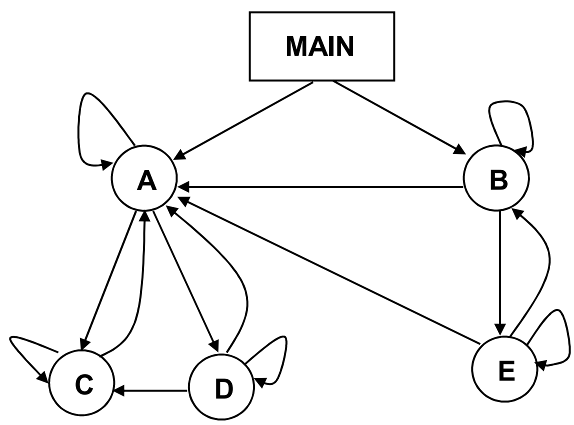
- Problems in static scoping
  – an errorneous call to a procedure that should not have been callable will not be detected as an error by the compiler
  – too much data access
  – hard to modify the structures of program safely



structure of program



desirable call graph



potential call graph

(2) Dynamic Scope

- *Dynamic scoping* is based on the calling sequence of subprograms (dynamic parent), not on their spatial relationship(static parent) to each other. Thus, the scope is determined at run time
- a convenient method of communication (parameter passing) between program units
- APL, SNOBOL4, some dialect of LISP
- Problems in dynamic scoping
  - the local variables of the subprogram are all visible to any other executing subprogram, regardless of its textual proximity
    - ⇔ results less reliable program
  - inability to statically type check referenced to nonlocals

5.9 Scope and Lifetime

- a variable declared in a Pascal procedure
  - scope : from its declaration to the end reserved word of the procedure (textual or spatial concept)
  - lifetime : the period of time beginning when the procedure is entered and ending when execution of the procedure reaches the end
- Example
  - scope of sum ?
  - lifetime of sum ?

```
procedure example ;
    procedure printerheader ;
      begin
        ....
      end ;
    procedure compute ;
      var sum : integer ;
      begin
        printerhead ;
      end
  begin
    compute;
  end ;
```


5.10 Referencing Environments

- The referencing environment of a statement is the collection of all names that are visible in the statement
  - static scoping languages
    - ⇔ the variable declared in its local scope + the collection of all variables of its ancestor scopes
  - dynamic scoping languages : ?
- Example

  Point 1 : x and y of sub1, a and b of example
  Point 2 : x of sub3, a and b of example
  Point 3 : x of sub2, a and b of example
  Point 4 : a and b of example

```
procedure example ;
    var a, b : integer ;
  procedure sub1 ;
      var x, y : integer ;
      begin
        ....          <- 1
      end ;
  procedure sub2
      var x : integer;
    procedure sub3 ;
        var x : integer ;
        begin
          .........   <- 2
        end
      begin
        ....          <- 3
      end
    begin
      ...             <- 4
    end ;
```

- active procedure : its execution has begun but has not yet terminated
- hidden variable : ?

5.11 Named Constant

- a named constant is a variable that is bound to a value only at the time it is bound to storage
- readability can be improved
  - Ex) `const listlen = 10` (Pascal)

5.12 Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization
  - In FORTRAN,

```
REAL PI
INTEGER SUM
DATA SUM /0/, PI /3.14159/
```

  - In Ada,
    - ⇔ `SUM : INTEGER := 0 ;`
  - In ALGOL68
    - ⇔ initializing variable : `int first := 10 ;`
    - ⇔ initializing named constant : `int second = 10 ;`

Homework
Problem Set : #2, #8, #9, #12, #13

<Homework>

1. Some programming languages are typeless. What are the obvious advantages and disadvantages of having no types in a language?
2. Consider the following JavaScript program: List all the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3, assuming static scoping is used.

```
var x, y, z;
function sub1() {
  var a, y, z;
  function sub2() {
    var a, b, z;
    . . .
  }
  . . .
}
function sub3() {
var a, x, w;
. . .
}
```

3. Consider the following Python program. List all the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3, assuming static scoping is used.

```
x = 1;
y = 3;
z = 5;
def sub1():
  a = 7;
  y = 9;
  z = 11;
. . .
def sub2():
  global x;
  a = 13;
  x = 15;
  w = 17;
  . . .
def sub3():
  nonlocal a;
  a = 19;
  b = 21;
  z = 23;
  . . .
  . . .
```

4. Consider the following skeletal C program. Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last function called? Include with each visible variable the name of the function in which it was defined.
   a. main calls fun1; fun1 calls fun2; fun2 calls fun3.
   b. main calls fun1; fun1 calls fun3.
   c. main calls fun2; fun2 calls fun3; fun3 calls fun1.
   d. main calls fun3; fun3 calls fun1.
   e. main calls fun1; fun1 calls fun3; fun3 calls fun2.
   f. main calls fun3; fun3 calls fun2; fun2 calls fun1.

```
void fun1(void); /* prototype */
void fun2(void); /* prototype */
void fun3(void); /* prototype */
void main() {
int a, b, c;
. . .
}
void fun1(void) {
  int b, c, d;
  . . .
}
void fun2(void) {
  int c, d, e;
  . . .
}
void fun3(void) {
  int d, e, f;
  . . .
}
```


5. Consider the following program, written in JavaScript-like syntax. Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last subprogram activated? Include with each visible variable the name of the unit where it is declared.
   a. main calls sub1; sub1 calls sub2; sub2 calls sub3.
   b. main calls sub1; sub1 calls sub3.
   c. main calls sub2; sub2 calls sub3; sub3 calls sub1.
   d. main calls sub3; sub3 calls sub1.
   e. main calls sub1; sub1 calls sub3; sub3 calls sub2.
   f. main calls sub3; sub3 calls sub2; sub2 calls sub1.

```
// main program
var x, y, z;
function sub1() {
  var a, y, z;
  . . .
}
function sub2() {
  var a, b, z;
  . . .
}
function sub3() {
  var a, x, w;
  . . .
}
```