Chapter 5

Names, Bindings, Type Checking, and Scopes

Chapter 5 Topics

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

Chapter 5

Names, Bindings, Type Checking, and Scopes

Introduction

- Imperative languages are abstractions of von Neumann architecture
  - Memory: stores both instructions and data
  - Processor: provides operations for modifying the contents of memory
- Variables characterized by attributes
  - Type: to design, must consider scope, lifetime, type checking, initialization, and type compatibility

Names

Design issues for names:

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

Name Forms

- A **name** is a string of characters used to identify some entity in a program.
- If too short, they cannot be connotative
- Language examples:
  - FORTRAN I: maximum 6
  - COBOL: maximum 30
  - FORTRAN 90 and ANSI C: maximum 31
  - Ada and Java: **no limit**, and all are significant
  - C++: **no limit**, but implementers often impose a length limitation because they do not want the **symbol table** in which identifiers are stored during compilation to be too large and also to simplify the maintenance of that table.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and (_).
- Although the use of the _ was widely used in the 70s and 80s, that practice is far less popular.
- C-based languages (C, C++, Java, and C#), replaced the _ by the “camel” notation, as in myStack.

- Prior to Fortran 90, the following two names are equivalent:

```
Sum Of Salaries  // names could have embedded spaces
SumOfSalaries    // which were ignored
```

- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
    - worse in C++ and Java because predefined names are mixed case (e.g. **IndexOutOfBoundsException**)
    - In C, however, exclusive use of lowercase for names.
  - C, C++, and Java names are case sensitive ➔ rose, Rose, ROSE are distinct names "What about Readability"

Special words

- An aid to readability; used to delimit or separate statement clauses
- A **keyword** is a word that is special only in certain contexts.
- Ex: Fortran

```
Real Apple       // Real is a data type followed with a
                 name, therefore Real is a keyword
Real = 3.4       // Real is a variable name
```

- **Disadvantage**: poor readability. Compilers and users must recognize the difference.
- A **reserved word** is a special word that **cannot** be used as a user-defined name.
- As a language design choice, reserved words are **better** than keywords.
- Ex: In Fortran, one could have the statements

```
Integer Real     // keyword "Integer" and variable "Real"
Real Integer     // keyword "Real" and variable "Integer"
```

Variables

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

Name

- Not all variables have names: **Anonymous**, heap-dynamic variables

Address

- The memory address with which it is associated
- A variable name may have different addresses at different places and at different times during execution.

  // `sum in sub1 and sub2`

- A variable may have **different** addresses at **different** times during execution. If a subprogram has a local var that is allocated from the run time **stack** when the subprogram is called, different calls may result in that var having different addresses.

  // `sum in sub1`

- The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement.

Aliases

- 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)

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.
- For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations for addition, subtraction, multiplication, division, and modulus.

Value

- The value of a variable is the contents of the memory cell or cells associated with the variable.
- Abstract memory cell - the physical cell or collection of cells associated with a variable.
- A variable's value is sometimes called its ***r*-value** because that is what is required when a variable appears in the **right** side of an assignment statement.

The Concept of Binding

- The ***l*-value** of a variable is its **address**.
- The ***r*-value** of a variable is its **value**.
- A **binding** is an association, such as 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:
  - *Language design time*: bind operator symbols to operations.
    - For example, the asterisk symbol (*) is bound to the multiplication operation.
  - *Language implementation time*:
    - A data type such as **int** in C is bound to a **range** of possible values.
  - *Compile time*: bind a variable to a **particular data type** at compile time.
  - *Load time*: bind a variable to a **memory cell** (ex. C **static** variables)
  - *Runtime*: bind a **nonstatic** local variable to a memory cell.

Binding of Attributes to Variables

- 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.

Type Bindings

- If static, the type may be specified by either an **explicit** or an **implicit** declaration.

Variable Declarations

- 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.)
- Both explicit and implicit declarations create static bindings to types.
- FORTRAN, PL/I, BASIC, and Perl provide implicit declarations.
- EX:
  - In **Fortran**, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention:
    I, J, K, L, M, or N or their lowercase versions is **implicitly** declared to be Integer type; otherwise, it is implicitly declared as Real type.
  - **Advantage**: writability.

- **Disadvantage**: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.
- In Fortran, vars that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.

- Less trouble with **Perl**: Names that begin with $ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure.
  - In this scenario, the names `@apple` and `%apple` are unrelated.
- In **C and C++**, one must distinguish between declarations and definitions.
  - **Declarations** specify types and other attributes but do **no** cause allocation of storage. Provides the type of a var defined external to a **function** that is used in the function.
  - **Definitions** specify attributes and cause storage allocation.

Dynamic Type Binding (JavaScript and PHP)

- Specified through an assignment statement
- Ex, JavaScript

```
list = [2, 4.33, 6, 8];     ➔ single-dimensioned array
list = 47;                  ➔ scalar variable
```

- – Advantage: **flexibility** (generic program units)
- – Disadvantages:
  - – **High cost** (dynamic type checking and interpretation)
    - Dynamic type bindings must be implemented using pure interpreter **not** compilers.
    - Pure interpretation typically takes at least **ten times** as long as to execute equivalent machine code.
  - – **Type error detection by the compiler is difficult** because **any** variable can be assigned a value of **any** type.
    - Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.
    - Ex:

```
i, x  ➔ Integer
y     ➔ floating-point array
i = x ➔ what the user meant to type
i = y ➔ what the user typed instead
```

    - **No error** is detected by the compiler or run-time system. i is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

Type Inference (ML, Miranda, and Haskell)

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

```
fun circumf(r) = 3.14159 * r * r;
      The argument and functional value are inferred to
      be real.

fun times10(x) = 10 * x;
      The argument and functional value are inferred to
      be int.
```

Storage Bindings & 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. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.
- Categories of variables by lifetimes: **static**, **stack-dynamic**, **explicit heap-dynamic**, and **implicit heap-dynamic**

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.
- **Advantages**:
    - **Efficiency**: (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocating and deallocating vars.
    - **History-sensitive**: have vars retain their values between separate executions of the subprogram.
- **Disadvantage**:
    - Storage **cannot** be shared among variables.
    - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

Stack-dynamic Variables:

- Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
- Ex:
    - The variable declarations that appear at the beginning of a **Java method** are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.
- Stack-dynamic variables are allocated from the **run-time stack**.
- If scalar, all attributes except address are statically bound.
- Ex:
    - Local variables in C subprograms and Java methods.
- **Advantages**:
    - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
    - In the absence of recursion it conserves storage b/c all subprograms share the same memory space for their locals.

- **Disadvantages**:
  - Overhead of allocation and deallocation.
  - Subprograms cannot be history sensitive.
  - Inefficient references (indirect addressing) is required b/c the place in the stack where a particular var will reside can only be determined during execution.
- In Java, C++, and C#, variables defined in **methods** are by **default** stack-dynamic.

Explicit Heap-dynamic Variables:

- Nameless memory cells that are allocated and deallocated by explicit directives “run-time instructions”, specified by the programmer, which take effect during execution.
- These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- The **heap** is a collection of storage cells whose organization is highly disorganized b/c of the unpredictability of its use.
- e.g. dynamic objects in C++ (via **new** and **delete**)

```
int *intnode;
…
intnode = new int; // allocates an int cell
…
delete intnode;  // deallocates the cell to which
                 // intnode points
```

- An explicit heap-dynamic variable of int type is created by the new operator.
- This operator can be referenced through the pointer, intnode.
- The var is deallocated by the **delete** operator.
- Java, all data except the primitive scalars are **objects**.
- Java objects are explicitly heap-dynamic and are accessed through **reference variables**.
- Java uses **implicit garbage collection**.
- Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.
- **Advantage**:
  - Provides for dynamic storage management.
- **Disadvantage**:
  - Inefficient “Cost of allocation and deallocation” and unreliable “difficulty of using pointer and reference variables correctly”

Implicit Heap-dynamic Variables:

- Bound to heap storage only when they are assigned value. Allocation and deallocation caused by **assignment statements**.

- All their attributes are bound every time they are assigned.
  - e.g. all variables in APL; all strings and arrays in Perl and JavaScript.
- **Advantage**:
  - Flexibility allowing generic code to be written.
- **Disadvantages**:
  - Inefficient, because all attributes are dynamic “run-time.”
  - Loss of error detection by the compiler.

Type Checking

- **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**.
- Ex: an **int** var and a **float** var are added in Java, the value of the **int** var is coerced to **float** and a floating-point is performed.
- A **type error** is the application of an operator to an operand of an inappropriate type.
- Ex: in C, if an **int** value was passed to a function that expected a **float** value, a type error would occur (compilers **didn't** check the types of parameters)
- If all type bindings are static, nearly all type checking can be static.
- If type bindings are dynamic, type checking must be dynamic and done at run-time.

Strong Typing

- A programming language is strongly typed if type errors are **always** detected. It requires that the types of all operands can be determined, either at compile time or run time.
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors.
- **Java and C#** are strongly typed. Types can be explicitly cast, which would result in type error. However, there are no implicit ways type errors can go undetected.
- The coercion rules of a language have an important effect on the value of type checking.
- Coercion results in a loss of part of the reason of strong typing – error detection.
- Ex:

```
int a, b;
float d;
a + d;    // the programmer meant a + b, however
```

– The compiler would not detect this error. Var a would be coerced to **float**.

Scope

- The scope of a var is the range of statements in which the var is visible.
- A var is **visible** in a statement if it can be referenced in that statement.
- **Local var** is local in a program unit or block if it is declared there.
- **Non-local var** of a program unit or block are those that are visible within the program unit or block but are not declared there.

Static Scope

- Binding names to non-local vars is called **static scoping**.
- There are two categories of static scoped languages:
    - Nested Subprograms.
    - Subprograms that can't be nested.
- Ada, and JavaScript allow **nested** subprogram, but the C-based languages do not.
- When a compiler for static-scoped language finds a reference to a var, the attributes of the var are determined by finding the statement in which it was declared.
- Ex: Suppose a reference is made to a var **x** in subprogram **Sub1**. The correct declaration is found by first searching the declarations of subprogram Sub1.
- If no declaration is found for the var there, the search continues in the declarations of the subprogram that declared subprogram Sub1, which is called its **static parent**.
- If a declaration of x is not found there, the search continues to the next larger enclosing unit (the unit that declared Sub1's parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success. ➔ an undeclared var error has been detected.
- The static parent of subprogram Sub1, and its static parent, and so forth up to and including the main program, are called the static **ancestors** of Sub1.

Ex: Ada procedure:

```
Procedure Big is
   X : Integer;
   Procedure Sub1 is
      Begin      -- of Sub1
      …X…
      end;       -- of Sub1
   Procedure Sub2 is
      X Integer;
      Begin      -- of Sub2
      …X…
      end;       -- of Sub2
   Begin         -- of Big
   …
   end;          -- of Big
```

- Under static scoping, the reference to the var X in Sub1 is to the X declared in the procedure Big.
- This is true b/c the search for X begins in the procedure in which the reference occurs, Sub1, but no declaration for X is found there.
- The search thus continues in the static parent of Sub1, Big, where the declaration of X is found.
- Ex: Skeletal C#

```
void sub()
{
  int count;
  …
  while (…)
  {
     int count;
     count ++;
     …
   }
   …
}
```

- The reference to count in the while loop is to that loop's local count. The count of sub is **hidden** from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a var with the same name in a larger enclosing scope.
- C++ and Ada allow access to these "hidden" variables
  - In Ada: Main.X
  - In C++: class_name::name

Blocks

- Allows a section of code to have its own local vars whose scope is minimized.
- Such vars are **stack dynamic**, so they have their storage allocated when the section is entered and deallocated when the section is exited.
- From ALGOL 60:
- Ex:

C and C++:

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

Ada:

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

Dynamic Scope

- The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic.
- Based on **calling sequences** of program units, not their textual layout (temporal versus spatial) and thus the scope is determined at **run time**.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- Ex:

```
Procedure Big is
   X : Integer;
   Procedure Sub1 is
      Begin      -- of Sub1
      …X…
      end;       -- of Sub1
   Procedure Sub2 is
      X Integer;
      Begin      -- of Sub2
      …X…
      end;       -- of Sub2
   Begin         -- of Big
   …
   end;          -- of Big
```

- Big calls Sub1
  - The dynamic parent of Sub1 is Big. The reference is to the X in **Big**.
- Big calls Sub2 and Sub2 calls Sub1
  - The search proceeds from the local procedure, Sub1, to its caller, **Sub2**, where a declaration of X is found.
- Note that **if static scoping** was used, in either calling sequence the reference to X in Sub1 would be to **Big's X**.

Scope and Lifetime

- Ex:
```
void printheader()
{
…
}     /* end of printheader */
void compute()
{
   int sum;
   …
   printheader();
}     /* end of compute */
```

- The **scope** of sum in contained within compute.
- The **lifetime** of sum extends over the time during which printheader executes.
- Whatever storage location sum is bound to before the call to printheader, that binding will continue during and after the execution of printheader.

Referencing environment

- It 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.
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to non-local vars in both static and dynamic scoped languages.
- 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.
- Ex, Ada, **static-scoped language**

```
procedure Example is
   A, B : Integer;
   …
   procedure Sub1 is
      X, Y : Integer;
      begin     -- of Sub1
      …                        ← 1
      end       -- of Sub1
   procedure Sub2 is
      X : Integer;
      …
      procedure Sub3 is
         X : Integer;
         begin  -- of Sub3
         …                     ← 2
         end;   -- of Sub3
    begin  -- of Sub2
    …                          ← 3
    end;   { Sub2}
begin
   …                           ← 4
end;       {Example}
```

- The referencing environments of the indicated program points are as follows:

Point	***Referencing Environment***
1	X and Y of Sub1, A & B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example
3	X of Sub2, A and B of Example
4	A and B of Example

- Ex, **dynamic-scoped language**
- Consider the following program; assume that the only function calls are the following: *main* calls *sub2*, which calls *sub1*

```
void sub1( )
{
  int a, b;
   …                 ← 1
}       /* end of sub1 */
void sub2( )
{
  int b, c;
   …                 ← 2
  sub1;
}       /* end of sub2 */
void main ( )
{
  int c, d;
   …                 ← 3
  sub2( );
}       /* end of main */
```

- The referencing environments of the indicated program points are as follows:

Point	***Referencing Environment***
1	a and b of sub1, c of sub2, d of main
2	b and c of sub2, d of main
3	c and d of main

Named Constants

- It is a var that is bound to a value only at the time it is bound to storage; its value **can't** be change by assignment or by an input statement.
- Ex, Java

```
final int LEN = 100;
```

- **Advantages**: readability and modifiability

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.
- Ex, Java

  int sum = 0;