Chapter 4

Variables:
Names, Bindings, Type Checking and Scope

Introduction

This chapter introduces the fundamental semantic issues of **variables**.

- It covers the nature of names and special words in programming languages, attributes of variables, concepts of binding and binding times.
- It investigates type checking, strong typing and type compatibility rules.
- At the end it discusses named constraints and variable initialization techniques.

Names

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

Case sensitivity

- Foo = Foo?
- The first languages only had upper case
- Case sensitivity was probably introduced by Unix and hence C.
- Disadvantage:
  - Poor readability, since names that look alike to a human are different; worse in Modula-2 because predefined names are mixed case (e.g. WriteCard)
- Advantages:
  - Larger namespace, ability to use case to signify classes of variables (e.g., make constants be in uppercase)
- C, C++, Java, and Modula-2 names are case sensitive but the names in many other languages are not

Special words

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

Def: 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 6-tuple of attributes:

Name: identifier
Address: memory location(s)
Value: particular value at a moment
Type: range of possible values
Lifetime: when the variable accessible
Scope: where in the program it can be accessed

Variables

- **Name** - not all variables have them (examples?)
- **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, but they are useful under certain circumstances

Aliases

- *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

Variables Type and Value

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

lvalue and rvalue

Are the two occurrences of "a" in this expression the same?

a := a + 1;

In a sense,

- The one on the *left* of the assignment refers to the location of the variable whose name is a;
- The one on the *right* of the assignment refers to the value of the variable whose name is a;

We sometimes speak of a variable's lvalue and rvalue

- The *lvalue* of a variable is its address
- The *rvalue* of a variable is its value

Binding

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:

- Language design time -- e.g., bind operator symbols to operations
- Language implementation time -- e.g., bind floating point type to a representation
- Compile time -- e.g., bind a variable to a type in C or Java
- Link time
- Load time--e.g., bind a FORTRAN 77 variable to memory cell (or a C static variable)
- Runtime -- e.g., bind a nonstatic local variable to a memory cell

Type Bindings

- *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 binding issues
  - How is a type specified?
  - When does the binding take place?
  - If static, type may be specified by either an explicit or an implicit declaration

Variable Declarations

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)

- E.g.: in Perl, variables of type scalar, array and hash begin with a $, @ or %, respectively.
- E.g.: In Fortran, variables beginning with I-N are assumed to be of type integer.
- E.g.: ML (and other languages) use sophisticated type inference mechanisms

Advantages: writability, convenience
Disadvantages: reliability

Dynamic Type Binding

- The type of a variable can chance during the course of the program and, in general, is re-determined on every assignment.
- Usually associated with languages first implemented via an interpreter rather than a compiler.
- Specified through an assignment statement, e.g. APL

```
LIST <- 2 4 6 8
LIST <- 17.3 23.5
```

- *Advantages:*
  - Flexibility
  - Obviates the need for "polymorphic" types
  - Development of generic functions (e.g. sort)
- *Disadvantages:*
  - High cost (dynamic type checking and interpretation)
  - Type error detection by the compiler is difficult

Type Inferencing

- Type Inferencing is used in some programming languages, including ML, Miranda, and Haskell.
- Types are determined from the context of the reference, rather than just by assignment statement.
- Legal:

```
fun circumf(r) = 3.14159 * r * r;   // infer r is real
fun time10(x) = 10 * x;             // infer r is integer
```

- Illegal:

```
fun square(x) = x * x;              // can't deduce anything
```

- Fixed

```
fun square(x) : int = x * x;        // use explicit declaration
```


Storage Bindings and Lifetime

- 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
- Categories of variables by lifetimes
  - Static
  - Stack dynamic
  - Explicit heap dynamic
  - Implicit heap dynamic

Static Variables

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

- Examples:
  - all FORTRAN 77 variables
  - C static variables

Advantage: efficiency (direct addressing), history-sensitive subprogram support
Disadvantage: lack of flexibility, no recursion!

Static Dynamic Variables

- Stack-dynamic variables -- 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
- *Advantages:*
  - allows recursion
  - conserves storage
- D*isadvantages:*
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)

Explicit heap-dynamic

Explicit heap-dynamic variables are 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
Example:

```
int *intnode;
. . .
intnode = new int;
. . .
delete intnode;
```


Implicit heap-dynamic

Implicit heap-dynamic variables -- Allocation and deallocation caused by assignment statements and types not determined until assignment.

e.g. all variables in APL
Advantage:
- flexibility

Disadvantages:
- Inefficient, because all attributes are dynamic
- 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
- Note:
  If all type bindings are static, nearly all checking can be static
  If type bindings are dynamic, type checking must be dynamic

Strong Typing

A programming language is *strongly typed* if

- type errors are always detected
- There is strict enforcement of type rules with no exceptions.
- All types are known at compile time, i.e. are statically bound.
- With variables that can store values of more than one type, incorrect type usage can be detected at run-time.
- Strong typing catches more errors at compile time than weak typing, resulting in fewer run-time exceptions.

Which languages have strong typing?

- Fortran 77 isn't because it doesn't check parameters and because of variable equivalence statements.
- The languages Ada, Java, and Haskell are strongly typed.
- Pascal is (almost) strongly typed, but variant records screw it up.
- C and C++ are sometimes described as strongly typed, but are perhaps better described as weakly typed because parameter type checking can be avoided and unions are not type checked
- Coercion rules strongly affect strong typing—they can weaken it considerably (C++ versus Ada)

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:
  - Subranges of integer types aren't compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

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

- More flexible, but harder to implement

Type Compatibility

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)

Type Compatibility 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:

Variable Scope

- The *scope* of a variable is the range of statements in a program over which it's visible
- Typical cases:
  - Explicitly declared => local variables
  - Explicitly passed to a subprogram => parameters
  - The *nonlocal* variables of a program unit are those that are visible but not declared.
  - Global variables => visible everywhere.
- The scope rules of a language determine how references to names are associated with variables.
- The two major schemes are **static** scoping and **dynamic** scoping

Static Scope

- Aka "lexical scope"
- Based on program text and can be determined prior to execution (e.g., at compile time)
- 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*

Blocks

- A block is a section of code in which local variables are allocated/deallocated at the start/end of the block.
- Provides a method of creating static scopes inside program units
- Introduced by ALGOL 60 and found in most PLs.
- Variables can be hidden from a unit by having a "closer" variable with same name

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

Examples of Blocks

C and C++:

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

Ada:

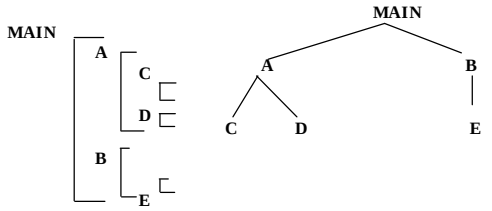
```
declare LCL :
 FLOAT;
  begin
  ...
  end
```

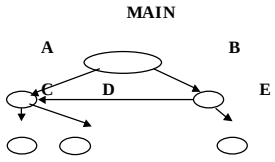
Common Lisp:

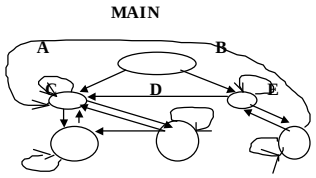
```
(let ((a 1)
      (b foo)
      (c))
   (setq a (* a a))
   (bar a b c))
```


Static scoping example

MAIN calls A and B
A calls C and D
B calls A and E







Evaluation of Static Scoping

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
- Used in APL, Snobol and LISP
  - Note that these languages were all (initially) implemented as interpreters rather than compilers.
- Consensus is that PLs with dynamic scoping leads to programs which are difficult to read and maintain.
  - Lisp switch to using static scoping as it's default circa 1980, though dynamic scoping is still possible as an option.

Static vs. dynamic scope

```
Define MAIN
   declare x
   Define SUB1
      declare x
      ...
      call SUB2
      ...

   Define SUB2
      ...
      reference 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 x

Dynamic Scoping

Evaluation of Dynamic Scoping:

- *Advantage:* convenience
- *Disadvantage:* poor readability

Scope vs. Lifetime

- While these two issues seem related, they can differ
- In Pascal, the scope of a local variable and the lifetime of a local variable seem the same
- In C/C++, a local variable in a function might be declared static but its lifetime extends over the entire execution of the program and therefore, even though it is inaccessible, it is still in memory

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.
- The value of a named constant can't be changed while the program is running.
- 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
- *Advantages:* increased readability and modifiability without loss of efficiency

Example in Pascal

```
Procedure example;
  type a1[1..100] of integer;
       a2[1..100] of real;
  ...
  begin
  ...
  for I := 1 to 100 do
    begin ... end;
  ...
  for j := 1 to 100 do
    begin ... end;
  ...
  avg = sum div 100;
  ...
```

```
Procedure example;
  type const MAX 100;
       a1[1..MAX] of integer;
       a2[1..MAX] of real;
  ...
  begin
  ...
  for I := 1 to MAX do
    begin ... end;
  ...
  for j := 1 to MAX do
    begin ... end;
  ...
  avg = sum div MAX;
  ...
```


Variable Initialization

- For convenience, variable initialization can occur prior to execution
- FORTRAN: Integer Sum
  Data Sum /0/
- Ada: Sum : Integer :=0;
- ALGOL 68: int first := 10;
- Java: int num = 5;
- LISP (Let (x y (z 10) (sum 0)) ...)

Summary

In this chapter, we see the following concepts being described

- Variable Naming, Aliases
- Binding and Lifetimes
- Type variables
- Scoping
- Referencing environments
- Named Constants
- Type Compatibility Rules