Variables:

Names, Bindings, Type Checking and Scope

Introduction

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.**

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
C++: ???

Case sensitivity

- Foo = foo?
- The first languages only had upper case
- Case sensitivity was probably introduced by Unix and hence C.

- **Disadvantage:**
  - **Poor readability**
  - **Worse 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

A *keyword* is **a word that is special only in certain contexts**

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**

Variables

- **A variable may have different addresses at different times during execution**

- **A variable may have different addresses at different places in a program**

Variables

- **If two (or more) 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**

- Some of the original justifications for aliases are no longer valid; e.g. memory reuse in FORTRAN

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**

Value - the contents of the location with which the variable is associated

lvalue and rvalue

Are the two occurrences of “a” in this expression the same?

a := a + 1;

lvalue and rvalue

Are the two occurrences of “a” in this expression the same?

a := a + 1;

- 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;

lvalue and rvalue

Are the two occurrences of “a” in this expression the same?

a := a + 1;

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

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**.

Binding

Possible binding times:

- Language design time -- e.g., bind operator symbols to operations
- 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

- A **binding is *static***
  if it **occurs before run time and remains unchanged throughout program execution.**

- A **binding is *dynamic***
  if it **occurs during execution or can change during execution of the program**.

Type Bindings

- **Type binding issues**
  - How is a type specified?
  - When does the binding take place?
  - Explicit or an implicit declaration

Static Type Binding

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

- E.g.: in Perl, variables of type array **begin with a $, @ or %, respectively.**
- E.g.: In Fortran, **variables beginning with I-N are assumed to be of type integer.**

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
```

Dynamic Type Binding

- ***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 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**

Type Inferencing

- **Legal:**

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

- **Illegal:**

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

- **Fixed**

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

Lifetime

- The ***lifetime*** of a variable **is the time during which it is combine to a particular memory cell**

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

Static Variables

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

- Examples:
  - C static variables

Static Variables

Advantage: efficiency (direct addressing), subprogram support

Disadvantage: no flexibility

Static Dynamic Variables

- Stack-dynamic variables are created for variables **when their declaration statements are built**
  - e.g. local variables in Pascal and C subprograms

Explicit heap-dynamic

Explicit heap-dynamic variables are **allocated** and **unallocated 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
- int *intnode;

  . . .

  intnode = new int;

  . . .

  delete intnode;

Explicit heap-dynamic

Explicit heap-dynamic variables are **allocated** and **unallocated by explicit directives**, specified by the programmer.

Advantage:
provides for dynamic storage management

Disadvantage:
uncontrollable

Implicit heap-dynamic

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

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 (tương thích) type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted (chuyển đổi ngầm), by compiler-generated code, to a legal type.

- A *type error* is the application of an operator to an operand of an inappropriate type

Strong Typing

A programming language is ***strongly typed*** if

- **type errors are always detected**
- **Applied of type rules with no exceptions.**
- **All types are known at compile time**
- **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,**

Which languages have strong typing?

- Fortran 77 isn't because it doesn't check parameters

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.

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

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 limit. Why ?

Type Compatibility

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

- More flexible, but harder to implement. Why ?

Type Compatibility

Consider the problem of two structured types.

- 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])
- ….

With structural type compatibility, you cannot differentiate between types of the same structure

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
  - Global variables => visible everywhere.

- The two major schemes are **static** scoping and **dynamic** scoping

Static Scope

- Also known as “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

Static Scope

- ***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/unallocated 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.

Examples of Blocks

C and C++:

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

Ada:

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

Common Lisp:

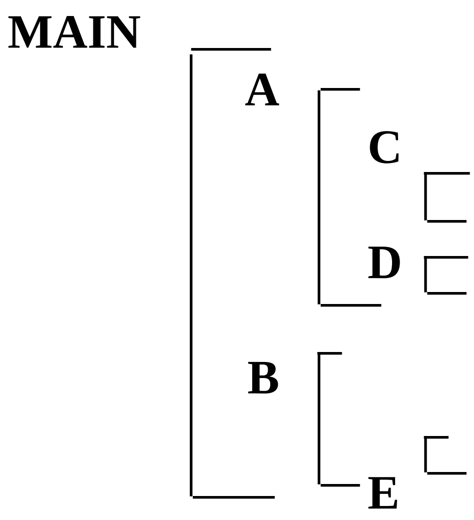
```
(let ((a 1)
      (b foo)
      (c))
   (setq a (* a a))
   (bar a b c))
```

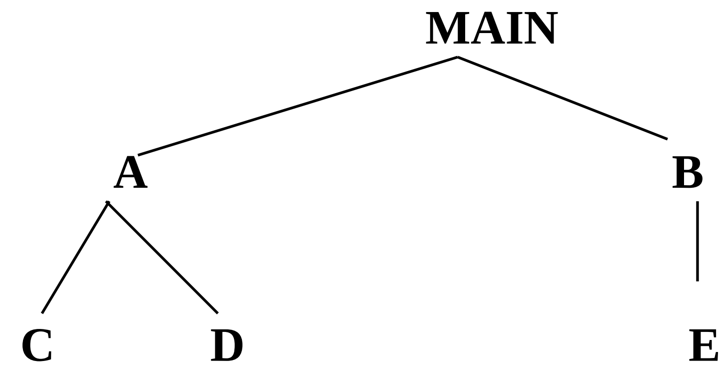
Static scoping example

MAIN calls A and B

A calls C and D

B calls A and E





Dynamic Scope

- Based on calling sequences of program units

- **References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point**

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

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**

Scope vs. Lifetime

- 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

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

Named Constants

- ***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;
  ...
```

Summary

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