Names, Bindings, Type Checking and Scopes

CS 315 – Programming Languages

Pinar Duygulu

Bilkent University

Introduction

- Imperative programming languages are abstraction of the underlying von Neumann computer architecture

- Memory : stores both instructions and data
- Processor: Provides operations for modifying the contents of the memory

Introduction

- Abstraction for memory – variables

- Sometimes abstraction is very close to characteristics of cells.
  - e.g. Integer – represented directly in one or more bytes of a memory
- In other cases, abstraction is far from the organization of memory.
  - e.g. Three dimensional array. Requires software mapping function to support the abstraction

Introduction

- A variable is characterized by a collection of properties (attributes)
  - Name
  - Address
  - Value
  - Type
  - Scope
  - Lifetime

Names

- Variables, subprograms, labels, user defined types, formal parameters have names.

- Design Issues
  - Maximum length of a name
  - Case sensitive or case insensitive
  - Special words (reserved words, keywords or predefined names)

Names

Name forms:

- Name is a string of characters

- Every language has a different string size.
  - Earliest languages : single character
  - Fortran 77 : up to 6 characters
  - Fortran 95 : 31 characters
  - C89 : no limit but only first 31 are significant
  - Java, C#, Ada : no limit
  - C++ : no limit , but sometimes implementors have

Names

Name forms:

- Names in most PL have the same form:
  - A letter followed by a string consisting of letters, digits, and underscore characters

- Today "camel" notation is more popular for C-based languages (e.g. myStack)

- In early versions of Fortran – embedded spaces were ignored. e.g. following two names are equivalent

  Sum Of Salaries

  SumOfSalaries

Names

Case sensitivity:

- In many languages (e.g. C-based languages) uppercase and lowercase letters in names are disctinct
  - e.g. rose, ROSE, Rose

- Problem for readability – names look very similar denote different entities

- Also bad for writability since programmer has to remember the correct cases
  - e.g. Java method `parseInt` for converting a string into integer, not `ParseInt` or `parseint`

Names

Case sensitivity:

- In C the problem can be avoided by exlusive use of lowercase letters for names

- In Java and C#, many of the predefined names include both uppercase and lowercase letters, so the problem cannot be escaped

- In Fortran 90, lowercase letters are allowed, and they simply translated to uppercase letters

Names

Special words:

- **reserved words** (special words that cannot be used as names):
- Can't define `for` or `while` as function or variable names.
- Good design choice

Names

Special words:

Keywords (special only in certain context):

- In FORTRAN, if `REAL` is in the beginning of a statement and followed by a name, it is considered as a keyword for declaration.
- Examples

  REAL APPLE *declaration*

  REAL = 8.7 *assignment*

  or

  INTEGER REAL

  REAL INTEGER

  This is allowed but **not readable**.
- Special words and names are distinguished by content

Names

Special words:

- **Predefined names** (have predefined meanings, but can be redefined by the user):
- For example, built-in data type names in Pascal, such as `INTEGER`, normal input/output subprogram names, such as `readln`, `writeln`, are predefined.

Variables

- Abstraction of a computer memory cell or collection of cells
- It is not just a name for a memory location
- It has six attributes: name, address, value, type, lifetime, scope

Variables

1. Name

- Most variables are named (often referred as identifiers).
- Although nameless variables do exist (e.g. pointed variables).

Variables

2. Address

- Associated memory location

- It is possible that the same name refer to different locations
- in different parts of a program.
  - A program can have two subprograms `sub1` and `sub2` each of defines a local variable that use the same name. e.g. `sum`
- in different times.
  - For a variable declared in a recursive procedure, in diferent steps of recursion it refers to different locations.

Variables

Aliases

- Multiple identifiers reference the same address – more than one variable are used to access the same memory location
- Such identifier names are called aliases.
- Aliases are not good for readability because the value of a variable can be changed by an assignment to its another name.
- can be created explicitly
  - by `EQUIVALENCE` statement in FORTRAN
  - by union types in C and C++
  - by variant record in Pascal
  - by subprogram parameters
  - by pointer variables.

Variables

3. Type

- Determines
  - the range of values the variable can take, and
  - the set of operators that are defined for values of this type.

- For example `int` type in Java specifies a range of -2147483648 to 2147483647

Variables

4. Value

- Contents of the memory cell or cells associated with the variable
- (abstract memory cell instead of byte size memory cells)

- *l_value* ← *r_value* (assingment operation)
- l_value of a variable: address of the variable
- r_value of a variable: value of the variable

The concept of Binding

- Binding: assocoaition of
- **attribute ↔ entity** or **operation ↔ symbol**
- Time of binding: **binding time** (important in the semantics of PL's)

Binding times

- language design time
  - * is bound to the multiplication operationion,
  - pi=3.14159 in most PL's.
- language implementation time
  - A data type, such as **`int`** in C is bound to a range of possible values
- compile time
  - A Java variable is bound to its type.

Binding times

- link time
  - A call to the library subprogram is bound to the subprogram code.
- load time
  - A variable is bound to a specific memory location.
- run time
  - A variable is bound to a value through an assignment statement.
  - A local variable of a Pascal procedure is bound to a memory location.

Binding times

- Example:
- count = count + 5
- The type of count is bound at compile time
- The set of possible values of count is bound at compiler design time
- The meaning of the operator symbol + is bound at compile time, when the types of its operands have been determined
- The internal representation of the literal 5 is bound at compiler design time
- The value of count is bound at execution times with this statement

Binding of attributes to variables

- **Static**: if binding occurs before runtime and remains unchanged throughout the program execution.
- **Dynamic**: if binding occurs during runtime or can change in the course of program execution.

Type bindings

- Before a variable can be referenced in a program it must be bound to a data type.

- How the type is specified
- When the binding takes place

Variable declarations

- **explicit declaration** (by statement)
  - a statement in a program that lists variable names and specifies that they are a particular type

- **implicit** declaration (by first appearance)
  - Means of associating variables with types through default conventions, rather than declaration statements. First appearance of a variable name in a program constitutes its implicit declaration

- **Both creates static binding to types**

Variable declarations

- Most current PLs require explicit declarations of all variables
- Exceptions: Perl, Javascript, ML

- Early languages (Fortran, BASIC) have implicit declarations
- e.g. In Fortran, if not explicitly declared, an identifier starting with I,J,K,L,M,N are implicitly declared to integer, otherwise to real type

- **Implicit** declarations are **not good for reliability** and **writability** because misspelled identifier names cannot be detected by the compiler
- e.g. In Fortran variables that are accidentally left undeclared are given default types, and leads to errors that are difficult to diagnose

Variable declarations

- Some problems of implicit declarations can be avoided by requiring names for specific types to begin with a particular special characters
- e.g. In Perl
  - $apple : scalar
  - @apple: array
  - %apple: hash

Dynamic type binding

- Type of a variable is not specified by a declaration statement, nor it can be determined by the spelling of its name
- Type is bound when it is assigned a value by an assignment statement.
- **Advantage:** Allows programming flexibility.

  example languages : Javascript and PHP
- e.g. In JavaScript
  - `list = [10.2 5.1 0.0]`
    - *list is a single dimensioned array of length 3.*
  - `List = 73`
    - *list is a simple integer.*

Dynamic type binding

Disadvantage:

1. less realiable : compiler cannot check and enforce types.

- Example:
- Suppose `I` and `X` are integer variables, and `Y` is a floating-point.
- The correct statement is
- I := X
- But by a typing error
- I := Y
- is typed. In a dynamic type binding language, this error cannot be detected by the compiler. `I` is changed to float during execution.
- The value of `I` becomes erroneous.

Dynamic type binding

Disadvantage:

2. cost :

- Type checking must be done at run-time.
- Every variable must have a descriptor to maintain current type.
- The correct code for evaluating an expression must be determined during execution.

- Languages that use dynamic type bindings are usually implemented as interpreters (LISP is such a language).

Type Inference

- ML is a PL that supports both functional and imperative programming
- In ML, the type of an expression and a variable can be determined by the type of a constant in the expression without requiring the programmer to specify the types of the variables
- **Examples**
  - **fun** circum (r) = 3.14 *r*r; (circum is real)
  - **fun** times10 (x) = 10*x; (times10 is integer)

- Note: **`fun`** is for function declaration.

Type Inference

- **fun** square (x) = x*x;
  - Default is int. if called with square(2.75) it would cause an error

- **It could be rewritten as:**
- **fun** square (x: real) = x*x;
- **fun** square (x):real = x*x;
- **fun** square (x) = (x:real)*x;
- **fun** square (x) = x*(x:real);

Storage Bindings and Lifetime

- **Allocation:** process of taking the memory cell to which a variable is bound from a pool of available memory
- **Deallocation:** process of placing the memory cell that has been unbound from a variable back into the pool of available memory
- **Lifetime of a variable:** Time during the variable is bound to a specific memory location
- According to their lifetimes, variables can be separated into four categories:
  - **static,**
  - **stack-dynamic,**
  - **explicit heap-dynamic,**
  - **implicit dynamic.**

Storage Bindings and Lifetime

Static Variables

- Static variables are bound to memory cells before execution begins, and remains bound to the same memory cells until execution terminates.
- **Applications:** globally accessible variables, to make some variables of subprograms to retain values between separate execution of the subprogram
- Such variables are **history sensitive**.
- **Advantage**: Efficiency. Direct addressing (no run-time overhead for allocation and deallocation).
- **Disadvantage**: Reduced flexibility.
- If a PL has only static variables, it **cannot** support **recursion**.
- Examples: All variables in FORTRAN I, II, and IV
- Static variables in C, C++ and Java

Storage Bindings and Lifetime

Stack-Dynamic Variables

- **Storage binding**: when declaration statement is elaborated (in run-time).
- **Type binding:** statical.
- **Example:** A Pascal procedure consists of a declaration section and a code section. The local variables get their type binding statically at compile time, but their storage binding takes place when that procedure is called. Storage is deallocated when the procedure returns.,
- Local variables in C functions.
- Advantages: Dynamic storage allocation is needed for recursion. Same memory cells can be used for different variables (efficiency)
- Disadvantages: Runtime overhead for allocation and deallocation

- In C and C++, local variables are, by default, stack-dynamic, but can be made static through **static** qualifier.

```
foo ()
{
static int x;
...}
```

Storage Bindings and Lifetime

Explicit Heap-Dynamic Variables

- Nameless variables
- storage allocated/deallocated by explicit run-time instructions
- can be referenced only through pointer variables
- types can be determined at run-time
- storage is allocated when created explicitly
- **Advantages:** Required for dynamic structures (e.g., linked lists, trees)
- **Disadvantages:** Difficult to use correctly, costly to refer, allocate, deallocate.

Storage Bindings and Lifetime

Implicit Heap-Dynamic Variables

- **Storage** and **type** bindings are done **when** they are **assigned values**.
- **Advantages:** Highest degree of flexibility
- **Disadvantages:**
- Runtime overhead for allocation and deallocation
- Loss of error detection by compiler
- Examples: Javascript and APL variables.

Type Checking

- Activity of ensuring that the operands of an operator are of compatible types.
- subprograms are also operators and parameters of subprograms are operands
- A type is **compatible** if it is legal for the operator or it can be converted to a legal type.
- The automatic type conversion is called **coercion**.
  - e.g. In addition of `int` variable with a `float` variable in Java `int` variable is coerced into `float` and floating point addition is done

Type Checking

- If type binding is static then all type checking can be done statically by compiler.
- Dynamic type binding requires dynamic type checking at run time, e.g. Javascript and PHP
- It is better to detect errors at compile time than at run time because the earlier correction is usually less costly
- However, static checking reduces flexibility
- If a memory cell stores values of different types (Ada variant records, Fortran Equivalance, C and C++ unions) then type checking must be done dynamically at run time
- So, even though all variables are statically bound to types in languages such as C++, not all type errors can be detected by static type checking

Strong typing

- A PL is a **strongly typed** language **if**
  - each name has a single type, and
  - type is known at compile-time.
- That is, **all types are statically bound**.

- A better definition:
- A PL is strongly typed if **type errors are always detected** (compile time or run time).
- allow functions for which parameters are not type checked.

Strong typing

- Examples:
- **FORTRAN77** is not strongly typed because
  - relationship between actual and formal parameters are not type checked.
  - EQUIVALANCE can be declared between different typed names.
- **PASCAL** is nearly strongly typed
  - except variant records because they allow omission of the tag field
- **Modula-2** is not strongly typed because of variant records.
- **Ada** is nearly strongly typed
  - variant records are handled better than PASCAL and Modula-2
- **C, ANSI C, C++** are not strongly typed
  - allow functions for which parameters are not type checked.

Strong typing

- Coercion weakens the value of strong typing
- Example:
- in Java the value of an integer operand is coerced to floating point and a floating operation takes place

- Assume that a and b are int variables. User intended to type a+b but mistakenly typed a + d where d is a float value. Then the error would not be detected since a would be coerced into float

Type compatibility

- The most important result of two variables being compatible types is that either one can have its value assigned to the other

- Two methods for checking type compatibility:
  - Name Type Compatibility
  - Structure Type Compatibility

Type compatibility

- **Name Type Compatibility:**
  - two variables have compatible types only if they are in either the same declaration or in declarations that use the same type name.
- Adv: Easy to implement
- Disadv: highly restrictive

Type compatibility

- **Name Type Compatibility:**
- under a strict interpretation a variable whose type is a subrange of the integers would not be compatible with an integer type variable

- Example:

```
type indexType = 1..10; {subrange type}
var count: integer;
index: indexType;
```

- The variables `count` and `index` are not name type compatible, and cannot be assigned to each other

Type compatibility

- **Name Type Compatibility:**
- Another problem arises when a structured type is passed among subprograms through parameters
- Such a type must be defined once globally
- A subprogram cannot state the type of such formal parameters in local terms (e.g. In Pascal)

Type compatibility

- **Structure Type Compatibility:**
- two variables have compatible types if their types have identical structure.
- Disadv: Difficult to implement
- Adv: more flexible

- The variables `count` and `index` in the previous example, are structure type compatible.

Type compatibility

- **Structure Type Compatibility:**
- Under name type compatibility only the two type names must be compared
- Under structure compatibility entire structures of the two types must be compared
- For structures that refer to its own type (e.g. linked lists) this comparison is difficult
- Also it is difficult to compare two structures, because
  - They may have different field names
  - There may be arrays with different ranges
  - There may be enumeration types

Type compatibility

- Structure type compatibility
- It also disallows differentiating between types with the same structure

  type celsius = float;

  fahrenheit = float;
- They are compatible according to structure type compatibility but they may be mixed

Type compatibility

- Most PL's use a **combination** of these methods.
- C uses structural equivalence for all types except structures.
- C++ uses name equivalence.

Type compatibility

- **Type compatibility (Ada)**
- Ada uses name compatibility
- But also provides two type constructs
  - Subtypes
  - Derived types

Type compatibility

- **type compatibility (Ada)**

- Derived types : a new type based on some previously defined type with which it is incompatible. They inherit all the properties of the parent type
- type celsius is new float
- type fahrenheit is new float
- Thee two types are incompatible, although their structures are identical
- They are also incompatible with any other floating point type

Type compatibility

- **type compatibility (Ada)**
- Subtype: possibly range constrained version of an existing type. A subtype is compatible with parent type

- Subtype small_type is Integer range 0..99;
- Variales of small_type are compatible with integer variales

Type compatibility

- **Type compatibility (Ada)**
- For unconstrained array types structure type compatibility is used

- Type vector is array (Integer range<>) of integer
- Vector 1: vector(1..10)
- Vector 2:vector(11..20)
- These two objects are compatible even though they have different names and different subscript ranges
- Because for objects of unconstrained array types structure compatibility is used
- Both types are of type integer, and they bot have then elements, therefore they are compatible

Type compatibility

- **Type compatibility (Ada)**
- For constrained anonymous arrays

- A: array(1..10) of integer;
- B: array (1..10) of integer
- A and B are incompatible

- C,D: array(1..10) of integer
- C and D are incompatible

- Type list_10 is array(1..10) of integer
- C,D:list_10;
- C and D are compatible

Type compatibility

- **Type compatibility in C**
- C uses structure type compatibility for all types except structures and unions
- Every struct and union declaration creates a new type which is not compatible with any other type

- Note that typedef does not introduce any new type but it defines a new name

- C++ uses name equivalence

Scope

- **Scope of a variable**:
- the **range of statements** 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 scope

- Scope of variables can be determined statically
  - by looking at the program
  - prior to execution
- **First** defined in **ALGOL 60**.

- Based on program text
- 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

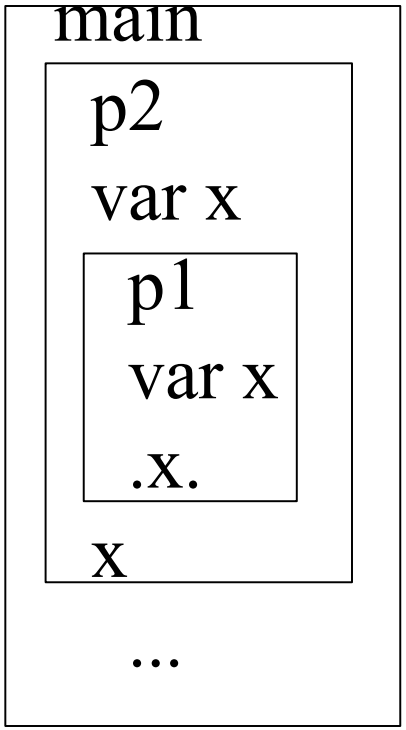
Static scope

- In all static-scoped languages (except C), procedures are nested inside the main program.
- Some languages also allow nested subprograms
  - Ada, Javascript, PHP - do
  - C based languages – do not
- In this case all procedures and the main unit create their scopes.

Static scope

- Enclosing static scopes (to a specific scope) are called its static ancestors;
- the nearest static ancestor is called a static parent

Static scope



main is the static parent of p2 and p1

P2 is the static parent of P1

Static scope

```
Procedure Big is
   x : integer
   procedure sub1 is
      begin   -   of
        sub1
      .... x ....
      end - of sub1
   procedure sub2 is
      x: integer;
      begin   -   of
        sub2
      ....
      end - of sub2
   begin - of big
   ...
   end - of big
```

The reference to variable x in sub1 is to the x declared in procedure Big

`x` in Big is hidded from `sub2` because there is another `x` in sub2

Static scope

- In some languages that use static scoping, regardless of whether nested subprograms are allowed, some variable declarations can be hidden from some other code segments
- e.g. In C++

```
void sub1() {
    int count;
    ...
    while (...) {
        int count;
        ...
    }
    ...
}
```

- The reference to count in while loop is local
- Count of sub is hidden from the code inside the while loop

Static scope

- Variables can be hidden from a unit by having a "closer" variable with the same name
- C++ and Ada allow access to these "hidden" variables
  - In Ada: **`unit.name`**
  - In C++: **`class_name::name`**

Blocks

- Some languages allow new static scopes to be defined without a name.
- It allows a section of code its own local variables whose scope is minimized.
- Such a section of code is called a block
- The variables are typically stack dynamic so they have their storage allocated when the section is entered and deallocated when the section is exited
- Blocks are first introduced in Algol 60

Blocks

- In Ada,

```
...

declare TEMP: integer;
   begin
   TEMP := FIRST;
   FISRT := SECOND;          Block
   SECOND := TEMP;
   end;
...
```

Blocks

C and C++ allow blocks.

```
int first, second;
...
first = 3; second = 5;
{ int temp;
      temp = first;
      first = second;
      second = temp;
}
...
```

temp is udefined here.

Blocks

- C++ allows variable definitions to appear anywhere in functions. The scope is from the definition statement to the end of the function

- In C, all data declarations (except the ones for blocks) must appear at the beginning of the function

- for statements in C++,Java and C# allow variable definitions in their initialization expression. The scope is restricted to the for construct

Dynamic scope

- APL, SNOBOL4, early dialects of LISP use dynamic scoping.
- COMMON LISP and Perl also allows dynamic scope but also uses static scoping

- In dynamic scoping
  - scope is based on the calling sequence of subprograms
  - not on the spatial relationships
  - scope is determined at run-time.

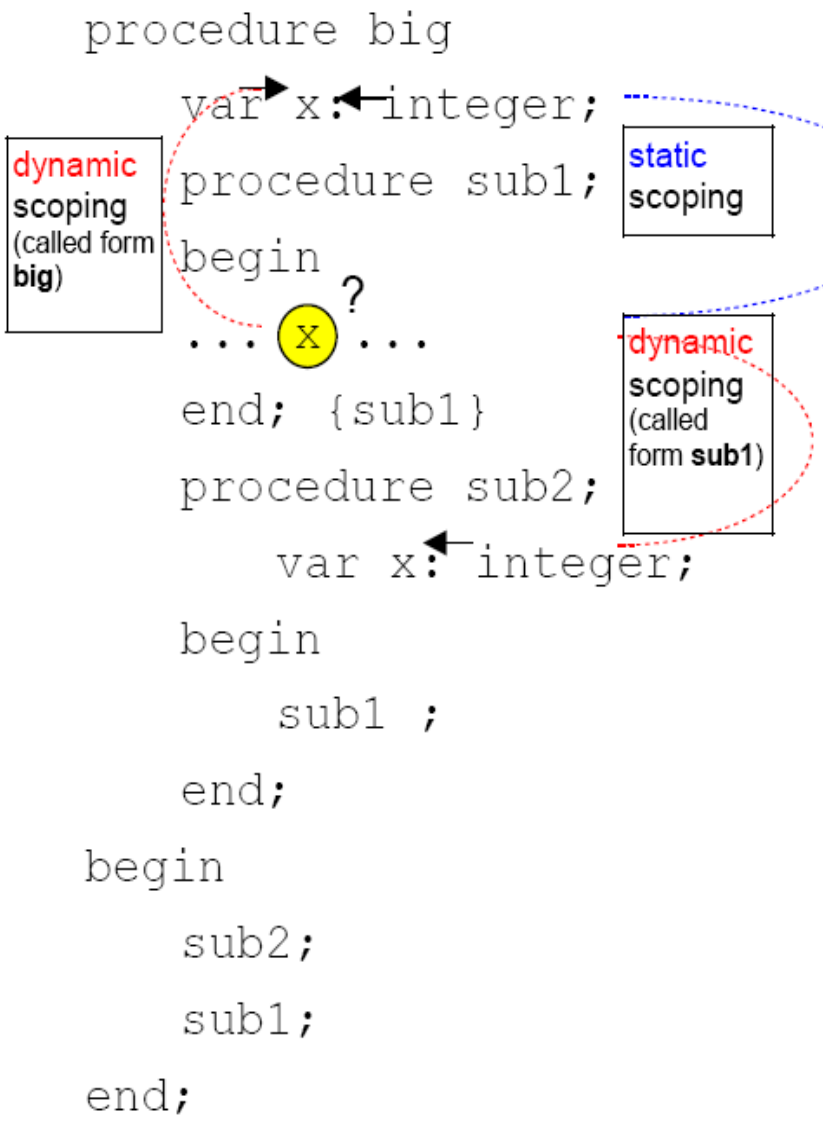
Dynamic scope

- When the search of a local declaration fails, the declarations of the dynamic parent is searched
- Dynamic parent is the calling procedure

```
Procedure Big is
    x : integer
    procedure sub1 is
        begin – of sub1
        .... x ....
        end – of sub1
    procedure sub2 is
        x: integer;
        begin – of sub2
        ....
        end – of sub2
    begin – of big
    ...
    end – of big
```

Big calls sub2
sub1 calls sub1
Dynamic parent of
sub1 is sub2
sub2 is Big

Dynamic scoping



To determine the correct meaning of a variable, first look at the local declarations.

For **static** or **dynamic scoping**, the **local variables are the same**.

In dynamic scoping, look at the dynamic parent (calling unit).

In static scoping, look at the static parent (unit that declares, encloses).

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, it is the local variables plus all of the visible variables in all of the enclosing scopes
- 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