2. 4 TYPE CHECKING

Type checking is the 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

- 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 Program Language 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).

It allows functions for which parameters are not type checked.

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.

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

Name 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

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

• 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)

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.

• 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

• 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

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

Type compatibility (Ada)

- Ada uses name compatibility
- But also provides two type constructs

– Subtypes

– Derived types

• 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

• 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 variables

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 both have then elements, therefore they are compatible

• 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 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

2.5 SCOPE

Scope of a variable is 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

Search process:

– search declarations,

•first locally,

•then in increasingly larger enclosing scopes,

•until one is found for the given name

In all static-scoped languages (except C), procedures are nested inside the main program.

• Some languages also allow nested subprograms

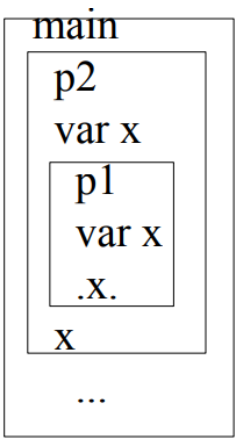
– Ada, Javascript, PHP - do

– C based languages – do not

• In this case all procedures and the main unit create their scopes.

Enclosing static scopes (to a specific scope) are called its static ancestors;

• The nearest static ancestor is called a static parent



main is the static parent of p2 and p1 p2 P2 is the static parent of P1

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

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

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

In Ada,

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

C and C++ allow blocks.

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

first = second;

second = temp;

}

...

temp is udefined here.

- 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.

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

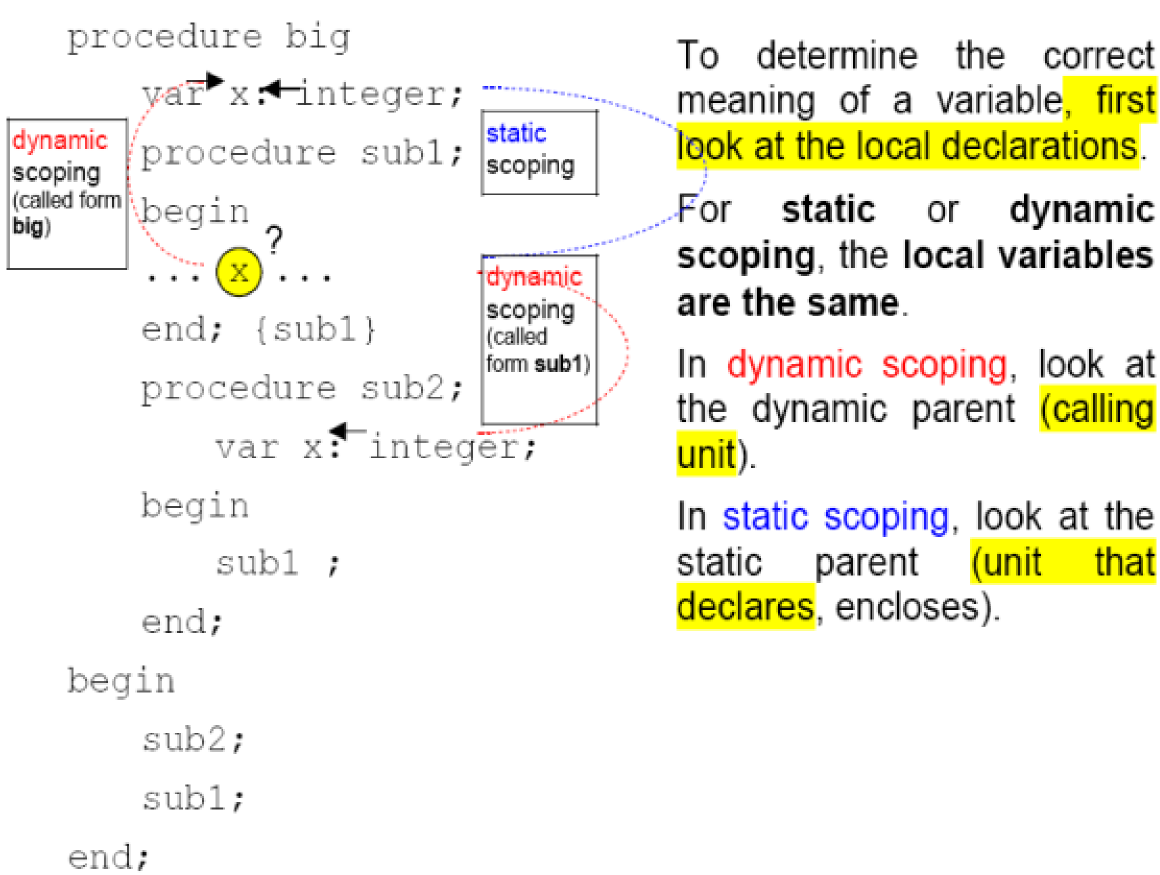
begin – of big

...

end – of big

Big calls sub2 sub1 calls sub1

Dynamic parent of sub1 is sub2 sub2 is Big



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