- **Names and referencing environments**

Two ways to make a data object available as an operand for an operation.

1. **Direct transmission** – A data object computed at one point as the result of an operation may be directly transmitted to another operation as an operand

   Example: x = y + 2*z;
   The result of multiplication is transmitted directly as an operand of the addition operation.

2. **Referencing through a named data object** –
   A data object may be given a name when it is created, and the name may then be used to designate it as an operand of an operation.

Program elements that may be named

1. Variables
2. Formal parameters
3. Subprograms
4. Defined types
5. Defined constants
6. Labels
7. Exception names
8. Primitive operations
9. Literal constants

Names from 4 thru 9 - resolved at translation time.
Names 1 thru 3 - discussed below.

Simple names: identifiers, e.g. *var1*.
Composite names: names for data structure components, e.g. *student[4].last_name*.

Associations and Referencing Environments

Association: binding identifiers to particular data objects and subprograms
Referencing environment: the set of identifier associations for a given subprogram.
Referencing operations during program execution: determine the particular data object or subprogram associated with an identifier.

Local referencing environment:

The set of associations created on entry to a subprogram that represent formal parameters, local variables, and subprograms defined only within that subprogram

Nonlocal referencing environment:

The set of associations for identifiers that may be used within a subprogram but that are not created on entry to it. Can be global or predefined.

Global referencing environment: associations created at the start of execution of the main program, available to be used in a subprogram,

Predefined referencing environments: predefined association in the language definition.

Visibility of associations

Associations are visible if they are part of the referencing environment. Otherwise associations are hidden

Dynamic scope of associations

The set of subprogram activations within which the association is visible

Aliases for data objects: Multiple names of a data object

- separate environments - no problem
- in a single referencing environment - called aliases.

Problems with aliasing

- Can make code difficult to understand for the programmer.
- Implementation difficulties at the optimization step - difficult to spot interdependent statements - not to reorder them

- **Static and dynamic scope**

The **dynamic scope of an association** for an identifier is that set of subprogram activations in which the association is visible during execution.

Dynamic scope rules

relate references with associations for names during program execution.

The **static scope of a declaration** is that part of the program text where a use of the identifier is a reference to that particular declaration of the identifier.

Static scope rules

relate references with declarations of names in the program text.

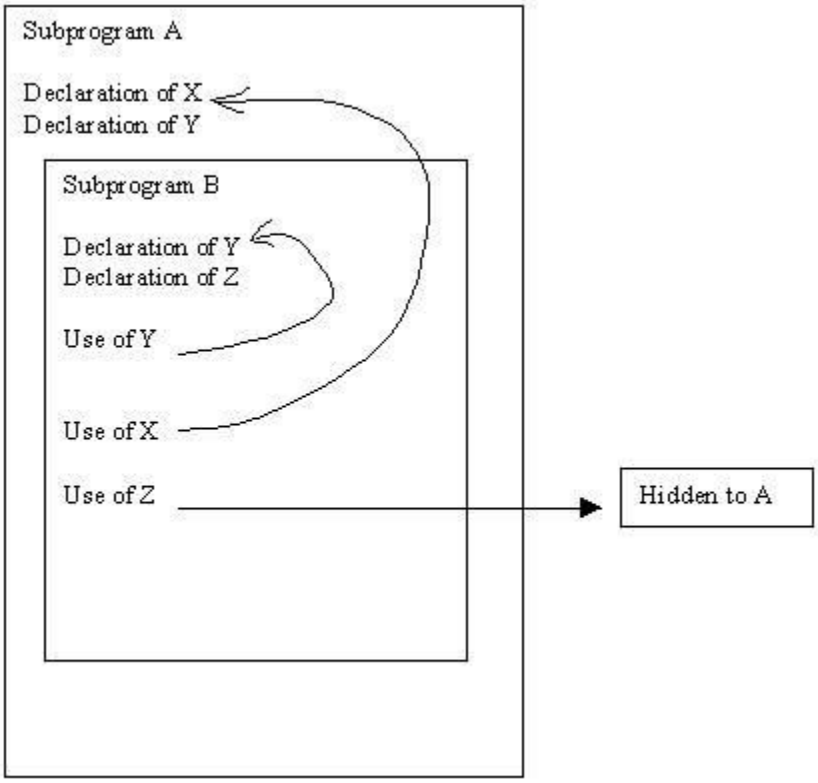
Importance of static scope rules - recording information about a variable during translation.

- **Block structure**

Block-structured languages (Pascal):

- Each program or subprogram is organized as a set of nested blocks.
- The chief characteristic of a block is that it introduces a new local referencing environment.

Static scope rules for block-structured programs



- **Local data and local referencing environments**

Local environment of a subprogram: various identifiers declared in the subprogram - variable names, parameters, subprogram names.

Static scope rules: implemented by means of a table of the local declarations
Dynamic scope rules: two methods:

- **Retention** - associations and the bound values are retained after execution.
- **Deletion** - associations are deleted.
  (For further explanation and example see Figure 9.9 on p. 369)

Implementation of dynamic scope rules in local referencing environments:
by means of a local environment table to associate names, types and values.

Retention: the table is kept as part of the code segment
Deletion: the table is kept as part of the activation record, destroyed after each execution.

- **Parameter transmission**

Subprograms need mechanisms to exchange data.

Arguments - data objects sent to a subprogram to be processed

Obtained through

- parameters
- non-local references

Results - data object or values delivered by the subprogram

Returned through

- parameters
- assignments to non-local variables
- explicit function values

1. **Actual and Formal Parameters**

A formal parameter is a particular kind of local data object within a subprogram.
It has a name, the declaration specifies its attributes.

An actual parameter is a data object that is shared with the caller subprogram.
Might be:

- a local data object belonging to the caller,
- a formal parameter of the caller,
- a nonlocal data object visible to the caller,
- a result returned by a function invoked by the caller and immediately transmitted to the called subprogram.

Establishing a Correspondence

Positional correspondence – pairing actual and formal parameters based on their respective positions in the actual- and formal- parameter lists.

Correspondence by explicit name – the name is paired explicitly by the caller.

2. **Methods for transmitting parameters**

Call by name – the actual parameter is substituted in the subprogram.

Call by reference – a pointer to the location of the data object is made available to the subprogram. The data object does not change position in memory.

Call by value – the value of the actual parameter is copied in the location of the formal parameter.

Call by value-result – same as call by value, however at the end of execution the result is copied into the actual parameter.

Call by constant value – if a parameter is transmitted by constant value, then no change in the value of the formal parameter is allowed during program execution.

Call by result – a parameter transmitted by result is used only to transmit a result back from a subprogram. The initial value of the actual-parameter data object makes no difference and cannot be used by the subprogram.

Note: Often "pass by" is used instead of "call by" .

Examples:

Pass by name in Algol

```
procedure S (el, k);
integer el, k;
begin
      k:=2; el := 0
end;
A[1] := A[2] := 1;
i := 1;
S(A[i],i);
```

Pass by reference in FORTRAN

```
SUBROUTINE S (EL, K)
K = 2
EL = 0
RETURN
END
A(1) = A(2) = 1
I = 1
CALL S (A(I), I)
```

Pass by name:

After calling `S(A[i],i)`, the effect is as if the procedure were

```
i := 2;
A[i] := 0;
```

As a result `A[2]` becomes 0.

On exit we have

`i = 2, A[1] = 1, A[2] = 0.`

Pass by reference:

Since at the time of call **i** is 1, the formal parameter **el** is linked to the address of `A(1)`.

Thus it is `A(1)` that becomes 0.

On exit we have: `i = 2, A(1) = 0, A(2) = 1`

3. **Transmission semantics**

Types of parameters:

input information
output information (the result)
both input and output

The three types can be accomplished by copying or using pass-by-reference

Return results:

Using parameters
Using functions with a return value

4. **Implementation of parameter transmission**

Implementing formal parameters:

Storage - in the activation record
Type:
Local data object of type T in case of pass by value, pass by value-result, pass by result
Local data object of type pointer to T in case of pass by reference

Call by name implementation: the formal parameters are subprograms that evaluate the actual parameters.

Actions for parameter transmission:

- associated with the point of call of the subprogram

each actual parameter is evaluated in the referencing environment of the calling program, and list of pointers is set up.

- associated with the entry and exit in the subprogram

on entry:
copying the entire contents of the actual parameter in the formal parameter, or copying the pointer to the actual parameter
on exit:
copying result values into actual parameters
or copying function values into registers

These actions are performed by *prologue* and *epilogue* code generated by the compiler and stored in the segment code part of the activation record of the subprogram.

Thus the compiler has two main tasks in the implementation of parameter transmission

3. It must generate the correct executable code for transmission of parameters, return of results, and each reference to a formal-parameter name.
4. It must perform the necessary static type checking to ensure that the type of each actual-parameter data object matches that declared for the corresponding formal parameter

- **Explicit common environment**

This method of sharing data objects is straightforward.

Specification: A common environment that is similar to a local environment, however it is not a part of any single subprogram.

It may contain: definitions of variables, constants, types.
It cannot contain: subprograms, formal parameters.

Implementation: as a separate block of memory storage.

Special keywords are used to specify variables to be shared.