Chapter 6
Semantics

<u>**Recall**</u> :

Programming Language **syntax** means what the language constructs look like.

Programming Language **semantics** means what those language constructs actually do (meaning).

Programming language semantics are much more complex to express than the syntax. Programming language semantics can be specified by :

1. The Programming language reference manual (most common and simple).
2. Translator (Compiler or Interpreter).
   By Experiment. Execute programs to find out what they do.
   Machine dependent (generally it is not portable).
3. Formal Definition (mathematical model). It is complex and abstract.

We will mainly be using the first method.

We will also use ALGOL-like languages in our discussion

Binding

Using names or identifiers in a programming language is a basic, fundamental abstraction - variable names, constant names, procedure and function names are all examples of this.

Related to names is the concept of:

location. Simply put, the location is the address of the name in memory. **Value**. Another thing related to the name is the **value**, which is the storable quantity in memory.

But how is the meaning of names determined?

Answer: It is determined by its **attributes** (properties associated with it).

For example :

const n = 15;

in this declaration, we associated 2 attributes :

1. It is a **constant** name.
2. it has a **value** of 15.

Another example :

var

x:integer;

again, 2 attributes are associated with the name x :

1. It is a **variable** name.
2. It is of **integer** type.

Another example :

```
function compute (n:integer, x:Real):Real;
Begin
    .
    .
end;
```

Associated with the name *compute* (function name) is :

1. It's type : a **function** name.
2. **Number and type** of parameters : it takes 2 parameters, one of type integer . and one of type Real .
3. It's **return value** : The function returns Real .
4. The **code body** of the function.

Another example :

var

y : ^ integer;

Associated with the name y is :

1. It's a **variable** name.
2. It is of a **pointer** to an **integer** type.

Notice that in all the examples above, all attributes are determined at **declaration**.

However, we can **assign attributes** outside the declaration.

For example :

x := 2;

this means that we add a **new attribute** to the name x , which is the **value**.

In the example,

var

y : ^ integer;

If we say,

New(y);

Then, in this case, we add a third attribute to the variable y which is the **location**.

When we first declared y, it pointed to junk (something random). When we used new(y) , pascal reserved a place in the memory the size of an integer and changed the reference to it (without having to name it, unlike C).

The process of associating attributes to names is called **Binding**. This happens at ***Binding Time***.

Binding Time : The time during the translation(compilation) process when the attribute is computed and associated to the name.

There are 2 kinds of binding times.

1. **<u>Static Binding</u>** : binding which occurs before execution. We call those attributes **static attributes**.
2. **<u>Dynamic Binding</u>** : binding which occurs during execution.

<u>Examples</u> :

1. *const n = 15;*

is a **static attribute**. This is because the attributes *<u>constant name</u>* and *<u>value=15</u>* are assigned during compilation.

2. x:integer

The attributes *<u>variable name</u>* & *<u>integer type</u>* are also **static attributes**.
However, when we say x:=2 , the attribute *<u>value=15</u>* is a **dynamic attribute** because it is assigned during **execution**.

3. y^:integer ;

the attributes *<u>variable name</u>* & *<u>pointer to integer</u>* type are **static attributes** , while new(y) , the added attribute *<u>location</u>* is a **dynamic attribute**.

Binding can be performed **<u>prior to translation</u>**.
As an examples in Pascal:

- Binding reserved words(names), Data structure types, and Array storage layout are predetermined at **language definition time**.
- Binding the values for the ***integer*** type (Range of integers) and the values for the ***Boolean*** type (true, false), The constant ***<u>maxint</u>*** in Pascal are defined at implementation time.

In short, Binding can be performed at :

- Language definition time.
- Language implementation time.
- Translation time.
  - at lexical analysis.
  - at syntax analysis.
  - at code generation.

All the above bindings are static.

- Execution time. This binding is dynamic.

Symbol Table

The Symbol Table is a special data structure used to maintain the binding during the translation process

Environment

The Environment is the memory allocation part of the execution process. ie, binding names to the storage locations is called Environment.

Memory

Memory is the binding the storage locations to values.

Declarations and Blocks

Declarations are the principle method to establish binding. There are 2 types of declarations:

1- **Explicit Declaration:**

Pascal:

```
Var
    X:integer;
    Ok:Boolean;
```

Algol68:

```
Begin
    Integer  x;
    Boolean ok;
End
```

Ada:

```
Declare
    X:integer;
    Ok:Boolean;
```

C:

```
Int x;
```

2- **Implicit Declaration:**

The variable is declared when it is used.

for example, in C: `int n = 10`

Declarations are associated with **blocks**. There are 2 types of blocks:

1- Main Program Block.

2- Procedure or function block.

For Example in Pascal:

```
Program Test;
  Var
    .
  Procedure P;
    Var
      .
   Begin
    .
    .
   End;

  function q:integer;
    Var
      .
   Begin
    .
    .
   End;

  Begin(*main*)
    .
    .
  End.
```

In Algol:

```
Begin
  Integer X;
  Boolean Y;
  .
  X := 2;
  Y ;= True
  .
End
```

In Ada:

```
Declare
  X : Integer;
  Y : Boolean;
  Begin
    X := 2;
    Y := 0;
  End;
```

Declarations bind different attributes to names especially the static type of attributes. **Note** that the declaration itself has an attribute, which is the position of the declaration in the program. This is important to determine the **scope/visibility** of the variable.

Scope of Declaration

The scope of declaration is :

The region of the program over which the declaration covers. In block structured languages, such as PASCAL , the scope of declaration is limited to the block in which is declared/appears and all other nested blocks. Contained within this block.

In fact, a language like PASCAL has the following scope rule :

RULE: The scope of declaration extends from the point it is declared to the end of the block.

for example :

```
Program scope;
  VAR X : Integer;

  Procedure P;
    VAR X:Real;
  BEGIN
       .
       .
  END;

  Procedure q;
   VAR Z:Boolean;
   BEGIN
       .
       .
  End;

     BEGIN(*main*)
        .
        .
     END.
```

In Algol 60:

```
A:BEGIN
  Integer X;
  Boolean Y;
  X:=2;
    .
    .
    .
  B:BEGIN
  Integer c,d;
  .
  .
  End;
  .
  .
  End
```

x, y have scope both in blocks A & B, while c, d have scope in block B only.

In Modula-2:

```
Module Ex;
   Procedure P;
       begin
           x:=2;
       end  P;
var
    x:integer;
begin
     .
     .
End Ex.
```

In Modula-2 the declaration extends all over the block backward & forward not just from the Point of declaration.

The scope of x extend all over the program block.

Important Note:

In block structured languages such as Pascal is that:

The declarations in nested blocks takes precedence over previous declarations.

Ex:

```
Program ex;
Var  x:integer;
Procedure P;
    Var   x:Real;  (*x  local to  P  *)
    Begin
        X:=3.5;
          .
    End;
Begin
     X :=2;   (*    x is the global   *)
      .
End.
```

That is, the global x can’t be accessed inside P, we say the “global X” has a scope hole inside P.

That is why we differentiate between ***scope*** and ***visibility***.

Visibility: The area where the name apply(excluding holes).

Scope: Including holes.

Symbol Table

All the declarations and binding are established by a structure called the ***symbol table***. In addition, the symbol table must maintain the *scope of declaration*. Different data structures can be used in the symbol table :

1. Hash Table --> static.
2. Linked List --> dynamic.
3. Tree Structure --> dynamic.

To maintain the scope of declarations correctly, the declarations should be processed using the stack concept(FILO). When entering a block, declarations are processed and attributes are added/bound to the symbol table (pushed to stack). When exiting from the block, the binding (the attributes) provided in the block are removed/popped from the stack.

Think of the symbol table as a **set of names**, each of which has a stack of declarations associated with it. The top of the stack is the current active declaration.

For example, consider the following pascal program :

```
Program symbol-table;
Var  X:integer;
     Y:boolean;
Procedure P;
    Var  X:boolean;
    Procedure  Q;
        Var   y:integer;
        Begin
           .
        End; (*Q*)
  Begin
      .
  End; (*P*)
Begin  (*main*)
    .
End.
```

There are 4 names declared in the program, X ,Y, P, Q

After Processing the global variables X, Y and procedure P, the symbol table looks like:

Boolean local to P		
Integer Global	Boolean Global	Procedure Global
X	**Y**	**P**

After processing procedure Q inside P ,

Boolean local to P	Integer local to Q		
Integer Global	Boolean Global	Procedure Global	Procedure local to P
X	**Y**	**P**	**Q**

After exiting from the body of procedure Q

Boolean local to P			
Integer Global	Boolean Global	Procedure Global	Procedure local to P
X	**Y**	**P**	**Q**

After exiting from the body of procedure P

Integer Global	Boolean Global	Procedure Global
X	**Y**	**P**

This scheme of scoping is called **Lexical scoping** or **Static Scoping**.

There are two types of scoping:

1- **Lexical (static)** scoping.
2- **Dynamic** scoping.

<u>Example:</u>

```
Program Scoping;
Var  X:integer;

Procedure P;

Begin
  Write(X);
End; (*P*)

Procedure  Q;
     Var   X:integer;
Begin
      X:=2;
      P;        .
End; (*Q*)

Begin  (*main*)
    X:=1;
    Q;
End.
```

- Now using the Lexical Scoping, the symbol table looks like:

Integer Global	Procedure Global
X	P

The value 1 is printed.

- Using Dynamic Scoping, the symbol table processes declarations as they are encountered in the ECXECUTION.

Integer local to Q		
Integer Global	Boolean Global	Procedure Global
X	P	Q

The value 2 is printed. Most block structured languages perform Lexical scoping. LISP Dynamic scoping.

Allocation and Environment

- Symbol table maintains in the declaration and the binding of attributes to names.

- Environment is binding names (or associating names) to locations.

- Environment may be constructed :

1. **Statically** (at load time) - Fortran.
2. **Dynamically** (at execution time) - Lisp.
3. **Mixture** (block structured languages such as Pascal, C, Modula-2, Ada, . . .)

Some allocations are performed statically and some dynamically.

Global variables: **<u>statically</u>**.

Local variables: ***<u>dynamically</u>***.

Some names are not bound to locations at all, for example:

const n=10;

The compiler replaces all occurrences of n by 10 in the block during execution with no need to allocate space for n.

- Environment in block structured languages binds locations to **local variables** in a **stack-based** fashion.

- During execution, on entering each block, the variables declared at the beginning of the block are allocated. On exit from that block, the same variables are deallocated.

-

<u>Example:</u>

```
Program Test;
Var  x, y : integer;

Procedure A(x:integer);
Var  y, z : real;

Begin
   .
End; (*A*)

Procedure  B(n:boolean);
      Var   y, z : real;

     Procedure C(h,p:real);
           Var  x,y : integer;
     Begin
           .
     End; (*C*)

Begin
    .
End; (*B*)

Begin  (*main*)
     .
     .
End.
```

Then the stack will look like:

After Entry to procedure A:

z
y
x
y
x

After Exit from A:

y
x

After Entry B & C:

y
x
p
h
z
y
n
y
x

Exit from C:

z
y
n
y
x

Generally, in block structured languages, there are 3 kinds of allocation in the Environment:

1) Static – Global variables.
2) Automatic – Local variables.
3) Dynamic – Pointers.

Static- Global variables
Automatic-Stack Local Variables
↓ **Both stack & heap grow in opposite direction** ↑
Dynamic- Heap Pointers