A Book Chapter

By Narendra Kumar Chahar

on **Static and Dynamic Scope, Block Structure**

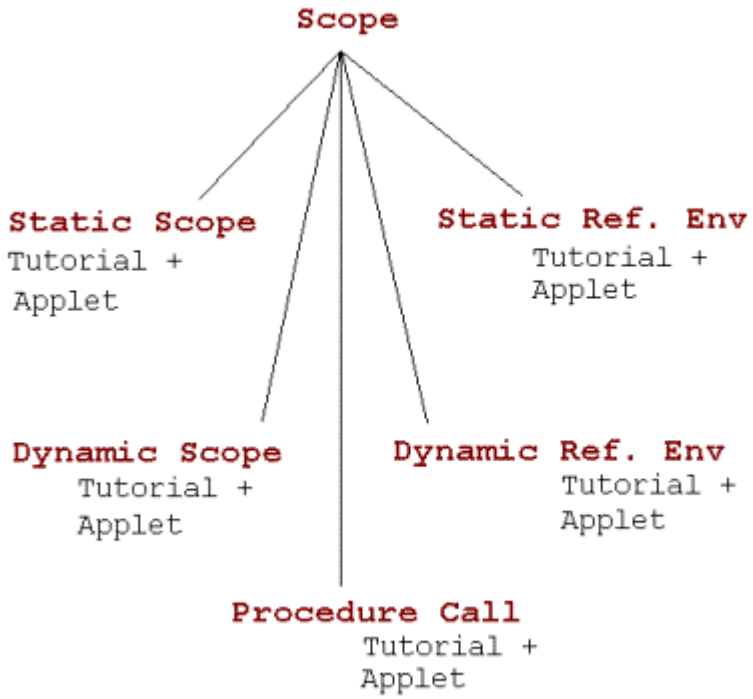
Index:

Chapter	Chapter Name	Page No.
8.	Static and Dynamic scope, Block structure	2 - 8

Chapter 8

Static and Dynamic scope

Scope is an important concept in programming languages – one cannot read or write large programs without properly understanding the concept of scope. The **scope** of a variable in a program is the lines of code in the program where the variable can be accessed.




```
// start pseudo-code
var y = "global";
function print-y() {
   print(y);
}
function test-scope() {
   var y = "local";
   print-y();
}
test-scope(); // statically scoped languages print "global"
              // dynamically languages print "local"

print-y(); // all languages should print "global"
// end pseudo-code
```

This is the standard type of example used to explain what static scoping is as compared to dynamic scoping. This makes sense to me, but never really sank in.

To anyone who already gets this, this will seem trivial. But the lightbulb went off for me when I thought about static vs dynamic typing…

In a dynamically typed language (like ruby, javascript, etc), types are not checked until execution. If an expression evaluates, then the type-checking worked. If not, it blows up to your error handling or the user. Statically typed languages check types at compile time. The programmer ensures that parameter types are specified and the compiler ensures the programmer's wishes will be followed.

Thinking in this fashion, static/dynamic scoping makes sense. For the following explanation, pretend that variables only have one type of storage for simplicity, and that global y is at memory location x01, while local y in test-scope is at x02.

If I'm a compiler in the act of compiling print-y (above code snippet) in a static language, then I know the scope I'm running in (hence static scope). I know that y is bound to the global variable, and I can replace that y with a direct location of x01 in the assembly I'm generating. No lookup tables, etc… fast.

If instead, I'm compiling print-y in a dynamic scope, then I can make no such substitution. I'm going to make some calls to print-y that will point to x01 and others that point to x02. What y is bound to be determined by the scope of the call at runtime… which is the definition of dynamic scoping?

So that might help it click. Everything said about a stack in dynamic scoping is true, but I think it's easier to understand that once you understand the above. Then you realize I could nest 4 or 5 of those calls and the last value of y would win.

Block Structure:

First the terminology used in the paper will be described. This includes the programming language notation used for describing the examples. Next the role of block structure will be discussed and finally we shall comment on some of the discussion of block structure in the literature.

In this paper block structure means textually nested procedures, classes and blocks as in Simula and Beta. The term object will be used as a common name for instances of procedures, classes and blocks.

By block is meant the Algol-60 type of statement. A block-activation is an instance of a block and covered by the term object.

The language used for describing the examples is restricted to a minimum. The following syntax is used:

<program> : : = <object-description>
<class-declaration> : : = <name> : class <object-description>
<procedure-declaration> : : = <name> :proc <formal-parameters> <object-description>
<formal-parameters> :: = (<input-parameters>) + (<output-parameters>)
<variable-declaration> : : = <name-list> : <object-specification>
<object-specification> : : = <class-name> 1 <object-description>
<object-description> : : = <super-class> begin <declaration-list> do <imperative-list> end
<super-class> :: = <class-name> 1 empty
<imperative> : : = <procedure-activation>
<object-description>
<variable-name> : = <expression>

A class-declaration and a corresponding variable-declaration may then appear as in the following program. Comments are enclosed by (and } .

```
begin
C :class S {S is the superclass of C}
begin (C-objects have 3 attributes}
Al : D; {Al is an instance of class D}
P :proc (X : integer, Y : boolean) + (2 : integer);
begin {I? is a procedure attribute}
{with two input parameters, X,Y}
{and one output parameter, Z.>
(X,Y, Z are themselves instance of classes)
end ;
T : class . . . (T is a class-attribute)
do I {I is an imperative that may be executed}
end ;
a : C; (a is an instance of class C, a C-object}
do . . .
end
```

In the example language variables have a type in the form of a class name. This means that the variable will denote an instance of that class or one of its subclasses. This is the same as in Simula. The generation time of objects has not been defined. An object may be generated together with the object containing the variable. Or objects may be generated by executing a new-imperative (like in Simula and Smalltalk). Both possibilities exist in Beta.

The example language also includes so-called singular objects, which are objects described directly without referring to a class or procedure:

```
begin {Bl}
F’ : begin { B2) I : integer end ;
do V.1 : = 7;
begin (B3)
X : integer
doX : = V.1; . . .
end ;
end
```

The whole program is a singular object described by Bl. The variable V is a singular object described by B2. B3 is a singular object describing an imperative in the form of an Algol-60 like block.

Most of the examples in this paper may (except for syntax) be expressed in the Beta programming language. Constructs not available in Beta are explicitly mentioned, some real or imaginary system, called the referent-system ([Delta]). In order to create a model of the referent-system concepts covering the relevant phenomena must be developed.

For a concept we shall use the classic terms which are: the name used to denote the concept, the intension: the properties of the phenomena covered by the concept, and the extension: the set of phenomena covered by the concept.

The model system (or program execution) contains elements corresponding to the phenomena and concepts selected as important for the desired perspective on the referent-system. Classes and procedures model concepts and objects model phenomena.

Abstraction mechanisms in programming languages are important. Most object oriented programming languages support the three fundamental sub functions of abstraction: classification, aggregation and generalization. The inverse functions exemplification, decomposition and specialization are similarly supported.
A class definition is a description of the intension of the instances (extension) of the class. This description includes: one or more superclasses specifying which classes/concepts that the new class specializes, a set of attributes characterizing instances of the new class, and an imperative-list that describes an action-sequence associated with instances of the class.

The attributes of a class/procedure may be described by referring to other classes/procedures, i.e. aggregation is taking place. The attributes may describe components that are a fixed part of the surrounding object, or components which are references to objects. Here block-structure or locality is important: Locality makes it possible to describe that an object is character by a concept in the form of a local

class or procedure. This restricts the existence of instances of such local classes or procedures to the lifetime of the enclosing object in which they are defined. In the remaining sections of this paper a number of examples of this will be given.

Block structure is not a mechanism for “programming in the large” in the sense that a program should be structured as a large program consisting of nested procedures and classes. A programming language must contain facilities for modularizing a program into minor parts. Especially aggregation should be supported by a construct like the Ada package allowing another hierarchy than block structure. In [BETA 83a] a language independent mechanism for program modularization is described.

A concept/abstraction is timeless in the sense that it has no state that changes over time. Since classes are used to model concepts, classes should not have state. An object is a phenomenon which has a state that may change over time. Objects may have the samefom, i.e. belong to the same class; but they have different substance. This means that they have a different location in terms of coordinates and time. Examples af objects are people, furniture, etc.

There are however phenomena which do not have substance ([Delta],[Beta]). A process (a partially ordered set of events) is an example of a phenomenon appearing in a program execution. The concepts covering such phenomena are typically modelled by procedures or concurrent process descriptions.

Values and types in programming languages model concepts where the phenomena are measurable properties of objects, i.e. the substance.

Discussion of Block Structure

There are many aspects of block structure being discussed in the literature. Here we shall comment on this discussion.

Locality: The major advantage of block structure is locality. This makes it possible to restrict the existence of an object and its description to the environment (object) where it has meaning.

Scope rules: There are (at least) the following aspects of scope rules for names declared within an object:

1. They only exist when the object exist. This is a consequence of locality.
2. Access to global names and redeclaration of names.

References:

[1]. Harper, R. (2016). *Practical foundations for programming languages*. Cambridge University Press.

[2]. Fernandes, E., & Kumar, A. N. (2004, March). A tutor on scope for the programming languages course. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 90-93).

[3]. Scott, M. L. (2000). *Programming language pragmatics*. Morgan Kaufmann.

[4]. Meijer, E., & Drayton, P. (2004, October). Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. OOPSLA.

[5]. Madsen, O. L. (1986, June). Block structure and object oriented languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming* (pp. 133-142).

[6]. Ferrari, A., Poggi, A., & Tomaiuolo, M. (2016). Object oriented puzzle programming. *Mondo Digitale*, *15*, 64.

[7]. Batteux, M., Prosvirnova, T., & Rauzy, A. (2018, October). From models of structures to structures of models. In *2018 IEEE International Systems Engineering Symposium (ISSE)* (pp. 1-7). IEEE.