PROGRAMMING LANGUAGES AND SUPPORT ENVIRONMENTS

J. N. Buxton
University of Warwick, United Kingdom

The Concept of Scope

The starting point for this consideration of programming language concepts is to look at two basic and early models of language - Fortran and Algol 60. These are languages of central historical importance: Fortran has been a major language since 1957 and is still the most widely used language for scientific processing and Algol is a direct ancestor of languages such as Pascal and Ada.

The scope of the definition of a named object in a program, such as a variable, can be simply defined as the area of the program text in which that object may be referenced and in which that name is valid. In very early languages scope was normally universal; that is, all definitions were valid throughout the program.

The first big step was the introduction of COMMON in Fortran II. Early Fortran compilers were prohibitively slow and this led to pressure for the separate compilation of program sections to avoid unnecessary and lengthy recompilation. Communication between such sections was provided in Fortran II in two ways. Control transfers and parameters were handled essentially by tagging relevant names as EXTERNAL and by a linkage loader system. Data references were handled by grouping data into COMMON areas whose constituent definitions were known throughout the program. The Fortran model of scope is essentially a two-level system of COMMON or global names and "within-subroutine" or local ones. This has remained essentially unchanged for over twenty years, though with some added flexibility in the introduction of multiple named common areas in more recent Fortrans. It gives a direct and practical solution to the requirement for separable compilation of large programs.

The Algol 60 model is markedly different. A program is divided into blocks and blocks may be nested within other blocks to any depth. Any block may contain definitions which are local to it and valid within it. In general, the definitions of surrounding or global blocks are also valid whereas definitions within inner blocks are inaccessible. The possibility clearly exists of redefinition of a name within an inner block - in this case, within any block the "nearest" or most local definition is valid and is regarded as shielding any more remote definitions using the same name.

The basically hierarchical view of a program as a nested structure of as many levels as necessary in depth fits very well with the view, that hierarchical decomposition is probably the only satisfactory way to order complexity. This point of view is very widely adopted by computer scientists. It does, however, lead to the serious practical difficulty that it is difficult to implement the requirement for program modularity through to the stage of separable compilation.
Furthermore, the design of data-flow oriented programs do not usually map easily or very naturally onto hierarchical structures and neither do problems exhibiting some forms of concurrency or with real time-dependent requirements. Another and less disturbing problem is that essentially low-level and detailed activities, typically implemented as commonly available subroutines, "float upwards" in the nested hierarchy so they are generally accessible throughout the program.

Substantial developments in the concept of scope have taken place on these early foundations, generally aimed at combining the best features of the practical simplicity of Fortran with the intellectual appeal of Algol. An important step in this development took place a decade or so ago with the introduction of specific lists of names to be declared as non-local to their defining module and hence accessible from elsewhere, together with corresponding lists of names defined elsewhere but required by a user module. The import-export list concept and other similar ideas are extensions of the EXTERNAL linkages in Fortran and allow highly specific control over visibility. This approach leads to network-like structures of program module linkages and interconnections, as contrasted to the simple two-level world of Fortran and the elegant hierarchical nesting of Algol.

A further and helpful step has been the introduction of name qualification to avoid ambiguities; usually by a dotted notation. If a name X is defined in two modules A and B, then in areas of the program where the meaning of X alone is ambiguous this can be resolved by using A.X or B.X as appropriate.

Concepts of Type

The second central concept we wish to discuss in this note is that of type. In the early languages, the concept was not clearly appreciated; the type of a variable was just a formalisation of what was representable and manipulable on the computer in question. Thus, the "basic types" generally available were integers, floating point numbers and so on and the operations available were those on the implementing machine.

This intuitive approach to type has now been replaced by clearer concepts. We now speak of a type as comprising:

(a) a range of values, represented in some internal and concealed way and

(b) a range of operations on these values which are useful and "safe", that is, they do not produce illegal results.

Clearly the use of types bears directly on the possible correctness of programs. If the variables of a program are partitioned into types, then illegal use of operators on operands of the wrong type can in principle be detected with consequent reductions in programming errors. The existence of a very few built-in types only in a language implies that this partitioning can only be at a very coarse level and this has motivated in more recent languages a search for ways to enable finer divisions into types.

The approach adopted is to allow, in various ways, the user to define his own types which can be tailored to best express the particular problem being addressed. The first steps in this direction were to introduce increasingly flexible data structures; arrays at first and then records, with components of already existing types. The language designs of the 1960's display many attempts to develop satisfactorily flexible and unified array and record structures.

More recently language designers have addressed the problem of allowing the user to introduce new types in more fundamental ways. We note two ideas which were both to some extent pioneered in Pascal: subrange types and enumerated types.The first of these presents no major conceptual difficulty: the user can introduce new "types" which share the representations and the operators of another and more basic type but which have fewer possible values, thus:

```
type shortinteger = integer (1...100);
variable a, b: shortinteger;
```

Variables of this new type can clearly hold values between 1 and 100 and have the other properties of integers.

Enumerated types are more confusing in nature in that the user here can introduce new "names" as data objects:

```
type colour = (red, yellow, blue, green);
variable paint, light : colour;
```

A confusion which can arise is that the name "red" is an internal data item; when first introduced in Pascal, for example, such a name could not be read or written as data. A name which is a member of an enumerated type list is a different sort of name from objects with which the programmer is familiar, such as variable or procedure names.

At the stage of language development typified by Pascal it is possible to move some way towards the idea of allowing the programmer to introduce new, e.g. structured, types together with operations upon them, though the latter may have to be described as procedures or functions. A simple and often-quoted example in Pascal is that of a rudimentary system for complex arithemetic which could be sketched as:

```
type complex = record
                 realpt: real, import: real        || definition
                 end                               || or "template"
var A, B, C : complex       ||instances of the "type"
proc Addcomp (i, j : complex)         ||an "operator"
    begin
       .
       .
       .
    end
```

This indeed enables us to provide objects of a user-defined type and to specify operations on values of the type. However, we have not succeeded in producing a fully satisfactory new type because we have failed to achieve safety in the representation. The programmer could accidently or deliberately use values of type "complex" for other purposes than those intended including, for example, accessing the components (or record fields) directly. The security of such a system depends only on convention, and there is no way analogous to type-matching with built-in types to ensure that only the appropriate operations are performed on these data items.

We are of course moving towards the modern concept of "abstract data types". The idea here is that a programmer should be able to specify new and perhaps quite complex classes of object in his program, together with appropriate operations, so as best to model his problem.
We repeat that the requirement is for language mechanisms such that:

(a) A type definition (or "template") can be made and instances of that type can be introduced,
(b) Operations on that type can be specified and their application can be checked by the supporting software, and
(c) The internal security of the representation of instances of such a type can be guaranteed.

To provide further examples we will consider the most popular source of examples of abstract data types - the stack. In a simple language we might write a "stack package" along the following lines:

```
begin
    MAX : constant: = 100;                  || the maximum length
    S : array (1 ... MAX) of integer        || so, we can only stack integers.
    TOP : (0 ... MAX)                       || i.e. a subrange definition
    procedure PUSH (X : integer) is         || an "operator"
    begin
    TOP : = TOP + 1;                        || (ommitting tests for
    S(TOP) : = X                            || overflow etc.)
    end
```

We have now the following problems:

(a) There is no way to access PUSH without also being able to access S and TOP, so the representation is not secure.
(b) We would really like to parameterise the length of the "stack type".
(c) And even also the type stacked
(d) And there is no initialisation mechanism for a new stack.

To proceed further towards a solution it will be necessary to return to consideration of questions of scope and visibility of names in a program in order to achieve security of representations.
The most widely available current approach to abstract data types is found in the Ada language. We use the stack example and Ada-like notation to illustrate the visibility control:

```
package STACK is                                    || The specification part,
    procedure PUSH (X : integer):                   || generally visible to
    function POP return integer;                    || "users" of the package,
end STACK                                           || i.e. to other modules
                                                    || requesting it.

package body STACK is                               || The body: names
    MAX : constant: = 100,                          || defined here and not
    S : array (1 ... MAX) of integer;               || in the specification
    TOP : integer range 0 ... MAX;                  || part are invisible to
    procedure PUSH is                               || users of the package.
       begin
         TOP = TOP + 1;
         S(TOP): = X;
    end PUSH;
(... similarly for POP etc..)
begin
    TOP: = 0                                        || initialisation
end STACK
```

This division of the program text of a module or package into public and private sections gives sufficient control on visibility. But it leaves unresolved the other inconveniences of the need for parameterisation of length and of the type stacked. Both of these can be handled in Ada by use of the "generic" features of the language. A generic definition may be very crudely regarded as akin to a macro-definition but with full security; it functions as a template for possibly many similar "instantiations" at compile time:

```
generic
    MAX : integer                                   ||the generic
    type ITEM is private                            ||"parameters"
package STACK is
    procedure PUSH (X : ITEM);
    function POP return ITEM;
end STACK

package body STACK is
    S : array (1 ... MAX) of ITEM;
    TOP : interger range 0 ... MAX;
    procedure PUSH is
       begin
         TOP: = TOP + 1;
         S(TOP): = X;
       end PUSH
    (... similarly for POP etc...)
begin
    TOP: = 0
end STACK

package STACKR is new STACK (100, real)             ||instantiations
package STACKINTS is new STACK (50, integer)
```

We do not attempt to describe this example in too great detail; for example, we prefer to gloss over the meaning of "private" above. The general idea is that having given a generic definition of a stack package we can cause many such packages to arise with systematic changes in the length and type stacked.

This now leads us to a further and interesting problem in visibility: the names PUSH and POP now are defined in two packages - viz STACKR and STACKINTS - which share the same visibility in the above example; how are they to be distinguished? There is an analogy here with polymorphic operations such as "+"; whether we mean "real+" or "integer+" is determinable by consideration of the types of the operands.

In the stack case, clearly

```
    PUSH (3.14)
```

must refer to the PUSH defined in STACKR. Suppose, however, we had also instantiated a package:

```
    LONGSTACKR is new STACK (5000, real)
```

then we would have to use some other way to differentiate which PUSH we mean. In this case we fall back on the qualified name device and refer to LONGSTACKR.PUSH or STACKR.PUSH as appropriate.

Programming Support Environments

The Ada language is an example of a large and complex modern programming language which provides both packages and block structures with various subtle devices for visibility control, together with user-defined types, generic definitions, multitasking and other wonders of the modern programming world. Yet nevertheless we still require some traditional facilities such as separable compilation. It is not really practicable to address the support of large programs written in such a language for use in the changing requirements of the real world by traditional software tools such as a classical operating system, compiler and link-loader. The compilation process itself, or instance, may need visibility over wide areas of program and may require resolution of many kinds of possible ambiguity.

Furthermore, a large application system is developed and maintained in many interconnecting modules over lifetimes of many years. During these years the individual modules will follow complex and different life histories, arising in many versions and joining in many configurations. All this history must be fully and completely tracked to enable enhancement and support of the system to continue throughout its economically useful life.

This is the world of the Programming Support Environments. The requirements for such a system for the support of Ada programs (the APSE) can be briefly summarised as:

(a) an integrated set of software tools covering the entire software lifecycle

(b) working around a project database with full historical recording.

Conclusion

It seems clear that the development of software engineering for at least the next few years will be much exercised with the development of APSE's and other large-scale support systems. Modern languages display much intellectual ingenuity in allowing the programmer to devise even more complex structures for his data and his algorithms and modern support systems will allow similar ingenuity in the structuring and version and configuration control of very large and long-lived programs. The concepts of scope and of type continue, however, to be central in understanding these developments.

Bibliography

Pascal

The source reference for Pascal which contains the original paper by Wirth is:
K. Jensen, N. Wirth
Pascal User Manual and Report
Springer Verlag, New York (1974)

There are many excellent Pascal textbooks of which two are
I.R. Wilson, A.M. Addyman
A Practical Introduction to Pascal
Springer Verlag, New York (1978);
P Grogono
Programming in Pascal
Addison-Wesley, 1980

A current source for the definition of revised, standard Pascal is
Specification for computer programming language Pascal
British Standards Institute BS6192 (1982).

Ada

The source reference for the language design by Ichbiah et al is
Reference manual for the Ada programming language
US Dept. of Defense (1980).
A somewhat revised version of the language is now in preparation. These are also many excellent textbooks, of which an example is
J.G.P. Barnes
Programming in Ada
Addison-Wesley, 1982.

Programming Environment

The source document for requirements for Ada Programming Support Environments (APSE's) is:
Buxton, J.N.
Requirements for Ada Programming Support Environments: "STONEMAN",
US Dept. of Defense, 1980.

This area is too recent to have led to textbooks and relevant work is accessible mainly in conference proceedings, for example in the AdaTEC Conferences on Ada organised by the ACM Special Interest Group.