The outer and inner syntax of a programming language

By M. V. Wilkes*

It is pointed out that the syntax of a higher level programming language such as ALGOL may be divided into two parts, to which the names outer and inner syntax are given. The outer syntax is concerned with the organisation of the flow of control, and is programmer-oriented, while the inner syntax is concerned with performing operations on data held in the memory of the computer, and is therefore data-oriented. It is shown how in the case of ALGOL it is possible to make a clean separation of the inner and outer syntax, and attention is drawn to certain practical advantages of regarding programming languages in this light.

(Received March 1968)

There are two sides to a programming language; one is concerned with organising the pattern of the calculation, and the other with performing the actual operations needed. These two sides are, in fact, nearly disjoint in ALGOL and similar languages, although this fact needs to be brought out. I hope that the discussion given in this note will provide some new insight into this aspect of programming languages. I must ask admirers of ALGOL to forgive me for taking that language as an example; my aim is to illustrate the formal problems involved in making a distinction between inner and outer syntax, and not to make any proposals relating to ALGOL itself.

Following the above approach, we may divide the syntax of a language into two parts. The first, referred to as the *outer* syntax, is concerned with performing the operations, and the second, referred to as the *inner* syntax, is concerned with organising the calculation. If the inner syntax is left undefined, then we have a language which can be called the outer language. This language is logically complete, but abstract in the sense that we do not commit ourselves in any way as to the nature of the objects declared, or define what the executable statements do. Having formally specified the outer language in this way, we are able to write a program to execute statements without having the least idea what the effect of that program will be. Bertrand Russell's definition of mathematics is worth quoting in this context: 'Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.'

In an outer language, we need a set of identifiers, a set of data types, a set of predicates, and a set of statements; typical members of these sets will be denoted by n_i, d_i, s_i, respectively. From the point of view of the outer language, these are all undefined entities. A program contains declarations by means of which identifiers are assigned to examples of particular data types, and it also contains statements taken from the set of which a typical member is s_i. The concept of an outer language will, perhaps, be clarified by the example given in **Table 1.** This is the outer syntax of a language based on ALGOL 60.

It will be seen that this syntax closely follows the relevant parts of the syntax of ALGOL 60 with the following exceptions.

(1) Features such as **comment** and dummy statement are omitted.
(2) The identifiers n_i are assumed to be available ready made and no recipe is given for constructing them out of letters. This is not a point of any importance.
(3) No **for** statements or **switch** declarations are defined. These are discussed below.
(4) No specific types are defined.
Instead of
⟨type⟩ ::= **real**|**integer**|**Boolean**
we have
⟨type⟩ ::= any one of d_i.
It follows that arithmetic expressions do not appear, although Boolean expressions are needed for conditional statements. Instead of
⟨Boolean primary⟩ ::= ⟨logical value⟩| ⟨variable⟩|⟨function designator⟩|⟨relation⟩| (⟨Boolean expression⟩)
we have
⟨Boolean primary⟩ ::= any one of p_i|(⟨Boolean expression⟩)

A language with this syntax is abstract in the sense in which that term is used above. A program can be written in the language but as long as the d_i, p_i, s_i remain undefined, the action of the program is also undefined.

If the d_i, p_i, s_i are now defined as in ALGOL 60, we have a fully defined language. The extra syntax that has been added is the inner syntax of this particular language. Note that the data types, and the data structures that represent them in the computer, are given concrete meanings when the inner syntax and associated semantics are defined.

Examples

To emphasise that the p_i and s_i do not necessarily

* *Director, University Mathematical Laboratory, Corn Exchange Street, Cambridge*

Table 1

Outer syntax of a language based on ALGOL 60

Delimiters

⟨delimiter⟩ ::=, | : | ; |(|)|**begin**|**end**|declarator

⟨declarator⟩ ::= any one of d_i| **procedure**

⟨logical operator⟩ ::= ≡ | ⊃ | ∨ | ∧ | ¬

Expressions

⟨expression⟩ ::= ⟨Boolean expression⟩|⟨designational expression⟩|

Boolean expressions

⟨Boolean primary⟩ ::= any one of p_i| (⟨Boolean expression⟩)

⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩| ¬ ⟨Boolean primary⟩

⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩|⟨Boolean factor⟩ ∧ ⟨Boolean secondary⟩

⟨Boolean term⟩ ::= ⟨Boolean factor⟩|⟨Boolean term⟩ ∨ ⟨Boolean factor⟩

⟨implication⟩ ::= ⟨Boolean term⟩|⟨implication⟩ ⊃ ⟨Boolean term⟩

⟨simple Boolean⟩ ::= ⟨implication⟩|⟨simple Boolean⟩ ≡ ⟨implication⟩

⟨Boolean expression⟩ ::= ⟨simple Boolean⟩|⟨if clause⟩ ⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

Designational expressions

⟨label⟩ ::= ⟨identifier⟩

⟨identifier⟩ ::= any one of n_i

⟨designational expression⟩ ::= ⟨label⟩|⟨if clause⟩ ⟨label⟩ **else** ⟨designational expression⟩

Compound statements and blocks

⟨unlabelled basic statement⟩ ::= any one of s_i|⟨goto statement⟩|⟨procedure statement⟩

⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩| ⟨label⟩ : ⟨basic statement⟩

⟨unconditional statement⟩ ::= ⟨basic statement⟩| ⟨compound statement⟩|⟨block⟩

⟨statement⟩ ::= ⟨unconditional statement⟩|⟨conditional statement⟩

⟨compound tail⟩ ::= ⟨statement⟩ **end** |⟨statement⟩; ⟨compound tail⟩

⟨block head⟩ ::= **begin** ⟨declaration⟩|⟨block head⟩; ⟨declaration⟩

⟨unlabelled compound⟩ ::= **begin** ⟨compound tail⟩

⟨unlabelled block⟩ ::= ⟨block head⟩; ⟨compound tail⟩

⟨compound statement⟩ ::= ⟨unlabelled compound⟩| ⟨label⟩ : ⟨compound statement⟩

⟨block⟩ ::= ⟨unlabelled block⟩|⟨label⟩ : ⟨block⟩

Goto statements

⟨go to statement⟩ ::= **goto** ⟨designational expression⟩

Conditional statements

⟨if clause⟩ ::= **if** ⟨Boolean expression⟩ **then**

⟨if statement⟩ ::= ⟨if clause⟩⟨unconditional statement⟩| ⟨label⟩ : ⟨if statement⟩

⟨conditional statement⟩ ::= ⟨if statement⟩|⟨if statement⟩ **else** ⟨statement⟩

Procedure statements

⟨procedure identifier⟩ ::= ⟨identifier⟩

⟨actual parameter⟩ ::= ⟨string⟩|⟨expression⟩|⟨procedure identifier⟩

⟨actual parameter list⟩ ::= ⟨actual parameter⟩|⟨actual parameter⟩, ⟨actual parameter⟩

⟨actual parameter part⟩ ::= ⟨empty⟩|(⟨actual parameter list⟩)

⟨procedure statement⟩ ::= ⟨procedure identifier⟩⟨actual parameter part⟩

Declarations

⟨type list⟩ ::= ⟨identifier⟩|⟨identifier⟩, ⟨type list⟩

⟨type⟩ ::= any one of d_i

⟨type declaration⟩ ::= ⟨type⟩⟨type list⟩

Procedure declarations

⟨formal parameter⟩ ::= ⟨identifier⟩

⟨formal parameter list⟩ ::= ⟨formal parameter⟩|⟨formal parameter⟩, ⟨formal parameter⟩

⟨formal parameter part⟩ ::= ⟨empty⟩| (⟨formal parameter list⟩)

⟨identifier list⟩ ::= ⟨identifier⟩|⟨identifier list⟩, ⟨identifier⟩

⟨value part⟩ ::= **value** ⟨identifier list⟩|⟨empty⟩

⟨specifier⟩ ::= ⟨type⟩|**label**|**procedure**|⟨type⟩ **procedure**

⟨specification part⟩ ::= ⟨empty⟩| ⟨specifier⟩⟨identifier list⟩; |⟨specification part⟩ ⟨specifier⟩⟨identifier list⟩;

⟨procedure heading⟩ ::= ⟨procedure identifier⟩ ⟨formal parameter part⟩; ⟨value part⟩⟨specification part⟩

⟨procedure body⟩ ::= ⟨statement⟩|⟨code⟩

⟨procedure declaration⟩ ::= **procedure** ⟨procedure heading⟩⟨procedure body⟩| ⟨type⟩ **procedure** ⟨procedure heading⟩⟨procedure body⟩

relate to numbers, I will take as examples the statements

```
load A
unload A
fire A
```

and the predicate

```
A is loaded
```

where *A* has been declared as the name of a gun by the declaration

```
gun A
```

An example of a compound statement is

```
begin  load A; fire A; end
```

A complete program block, containing the declaration of *A*, would be

```
begin  gun A;
       load A; fire A;
       unload A;
end
```

The statement *unload A* is strictly unnecessary, but is included as a precaution against untoward happenings during the debugging process.

An example of a conditional jump is given in the following program for reducing a fort. Note the new predicate *F flying* where it is assumed that *F* has been declared in an outer block to be of type **flag**.

```
begin  gun A
       L1: load A; fire A;
       if F flying goto L1;
end
```

In order that it should be possible to write programs, certain relations must exist between statements and predicates. With exceptions to be mentioned presently, there must be at least one statement (which can be a compound statement) such that, after the execution of that statement, a given predicate would, if tested, be found to be true; similarly, there must exist at least one statement (which may be compound) such that, after the execution of that statement, the negation of a given predicate would, if tested, be found to be true. Statements and predicates, or statements and negations of predicates, connected in this way may be said to be *associated*.

In the above example, the predicate *A is loaded* is associated with the statement *load A*, while *unload A* and *fire A* are both statements associated with *A is loaded*.

The exceptions refer to predicates depending on conditions outside the direct control of the program, for example, on the state of a peripheral device.

The for statement

The **for** statement is essentially a device for controlling the flow of the program and belongs properly, one would think, to the outer syntax. However, in ALGOL 60, **for** statements can contain expressions, the syntax of which belongs to the inner syntax. Some analysis must, therefore, be undertaken before a surgical operation can be performed which will make a clean break between the outer and the inner syntax.

In the outer syntax given above, the only completely defined statement is **goto**. An enrichment of the outer syntax is the inclusion of an assignment statement for identifiers. Such a statement would permit identifiers to appear to the right and left of the := sign. If this addition is made, it is possible to define within the outer syntax a **for** statement of the type regarded as primitive in ALGOL, that is, a **for** statement having an explicit **for** list.

An example of the use of such a statement is as follows:

```
begin
    gun X;
    for X := A, B, C do
        begin load X;
            fire X;
        end
end
```

The most useful type of **for** statement, however, is one that allows counting, and in order to introduce this into the outer syntax, we must specify one of the data types to be **integer**. Defining one of the data types in this way does not, as we shall see, seriously affect the abstract status of the outer language. It is now possible to write the following procedure for firing a salute.

```
procedure salute (n); value n;
begin integer t; gun A;
    for t := 1 step 1 until n do
        begin
            load A; fire A
        end
end
```

Thus when the head of a state approaches, the procedure call *salute* (21) would be in order.

Although, for the purpose of the **for** statement, a data type **integer** has been defined in the outer language, it is not necessary that this data type should be made use of when implementing the inner language. It is, of course, open to us to do this, since everything defined in the syntax of the outer language remains valid in the inner language. Circumstances in which it would be more convenient to have a different technique for representing integers in the inner language are:

(1) When the technique that is natural and efficient for representing integers in the outer language would not be appropriate in the inner language. This could be the case, for example, if integers were stored in the inner language in the manner used in LISP.

(2) When it is desired to have a standard outer language into which a number of separately written

inner languages can be 'plugged'. The task of implementing a given inner language is made more self-contained, although perhaps at the cost of a small amount of redundancy, if any data types that have been defined in the outer language are ignored.

If similar data types exist in both outer and inner languages, the nomenclature must, of course, be chosen to keep them separate. One could, for example, use **counter** in the outer syntax, and **integer** in the inner syntax.

The remarks made above about the **for** statement apply also to the **switch,** which also needs integers for its implementation.

We have now obtained a very clean (although not necessarily unique) cut between the outer and inner syntax. It is not, however, possible for one of the variables in the **for** statement—the range for example—to be changed by the program. We would, therefore, need to provide a transfer function which would enable an integer belonging to the outer syntax to be set equal to an integer belonging to the inner syntax. If the same data structure is used in both the inner syntax and the outer syntax, this transfer function is, of course, the identity operator. In other cases, a subroutine would have to be specially written when the inner syntax was being implemented. Naturally, in order to write this subroutine, a knowledge of the data representation used for storing integers in the outer language would be needed.

Conclusion

The primary purpose of this note will have been served if it has been made clear that there are two influences controlling the form of a programming language; these influences operate from opposite ends. There are features of the language controlled entirely by the predilections of the programmer, ranging from genuine minimum requirements as to what he is able to say, to fads and fancies. There are other features controlled by the nature of the data types and representations that the language is designed to manipulate, and the storage allocation scheme that is used to accommodate them; in this class, the most fundamental concern data types and the operations to be performed on the data.

It is not to be expected that any existing language will separate cleanly into the two parts. It has, however, been shown that, in the case of ALGOL, the separation can be effected without great cost. No doubt it could be effected in ways other than the one given here, since there is a certain amount of arbitrariness about what is regarded as belonging to the outer language and what is regarded as belonging to the inner language. In whatever way the separation is done, we shall be left with an outer language of great manipulative power, to which can be fitted a variety of inner languages, each designed to manipulate a different set of data structures. It is suggested that along these lines might be found a solution to the problem of enabling a sufficiently skilled programmer to extend a language by adding new data types while preserving intact the external form of the language.

Book Review

The Information Centre, by Morton F. Meltzer, 1968; 160 pages. (London: *Bailey Bros. and Swinfen Ltd.*, 85s. 6d.)

This well-written book presents to top management the case for establishing a comprehensive information centre to deal not only with purely scientific or technical information relevant to the business but one which also acquires, files and distributes, as needed, all relevant commercial information as well.

In presenting his case the author finds himself in a difficulty. Because he demands the critical attention of busy senior executives he cannot afford to be lengthy. But he also needs to display all the specialist techniques that the manager of a technical information centre can command (thereby also justifying the salary he claims), but convincing demonstrations of specialist skills rapidly get wordy or technically intricate, and therefore unreadable by executives. So the author has to compromise. The result is a readable report or essay of 126 pages (rather expensive at 8d. per page) which, to the uninitiated, teasingly mentions rather than explains many facets of information work.

So, in spite of its brevity, the book is comprehensive. The author helpfully admits that information services cost money and boldly produces some realistic estimates of the initial and running costs of an information department. His weakest chapter is on 'Determining the return on investment' on an information centre. It is unconvincing because no one has yet been able to quantify the return on information *per se*. Information is only one of many elements in the complex of human activity engaged in productive enterprises and is peculiarly resistant to isolation and objective quantification.

The greatest danger that faces the author is that the book could succeed too well. The executive who reads it might discern that the role assigned by the author to the manager of the technical information centre he describes would make the manager a mighty power in the business. The executive might decide that this book could best be regarded as an outline systems analysis of an information service which would more safely be provided by upgrading, not the manager of the technical information centre, but the firm's computer.

Foreword, acknowledgements, notes, glossary, selected bibliography and index all help to provide a grand total of 160 pages.

B. C. Brookes (London)