1. Introduction

1.1 General purpose programming languages

High level languages, like FORTRAN, ALGOL 60 and COBOL were originally regarded as useful for two purposes:

- to provide concepts and statements allowing a precise formal description of computing processes and also making communication between programmers easier.

- to provide the non-specialist with a tool making it possible for him to solve small and medium-sized problems without specialist help.

High level languages have succeeded in these respects. However, strong new support for these languages is developing from a fresh group: those who are confronted with the task of organizing and implementing very complex, highly interactive programs, e.g. large simulation programs.

These tasks put new requirements on a language:

- in order to decompose the problem into natural, easily conceived components, each part should be describable as an individual program. The language should provide for this and also contain means for describing the joint interactive execution of these sub-programs.

- in order to relate and operate a collection of programs, the language should have the necessary powerful list processing and sequencing capabilities.

- in order to reduce the already excessive amount of debugging trouble associated with present day methods, the language should give "reference security". That is, the language and its compiler should spot and not execute invalid use of data through data referencing based on wrong assumptions.

Even if the organizational aspects of complex programming are becoming more and more important, the computational aspects must, of course, be taken care of at least as well as in the current high-level languages.

It is also evident that such a general language should be oriented towards a very wide area of use. The market cannot for long accommodate the present proliferation of languages.

1.2 Special application languages

Until now, the computer has been a powerful but frightening tool to most people. This should be changed in the years to come, and the computer should be regarded as an obvious part of the human environment. More and more people should get their capabilities increased through the availability of the "know-how" and data they need.

A condition for this development is that the demands on the computer user are reduced, which implies that communication between man and computer is made easier.

Know-how is today to a large extent made operative through "application packages" covering various fields of knowledge and methods. But these packages are in general not sufficiently flexible and expandable, and also often require specialist assistance for their use.

The future seems to be "application languages" which are problem-oriented, perhaps in the extreme. Such languages may provide the basic concepts and methods associated with the field in question and allow the user to formulate his specific problem in accordance with his own earlier training.

At the same time, such languages should be flexible in the sense that new knowledge acquired should be easily incorporated, even by the individual user.

The need for application languages is apparently in conflict with the desire for the non-proliferation of languages and for general purpose programming languages.

A solution is to design a general purpose programming language to serve as a "substrate" for the application languages by making it easy to orient towards specialized fields, and to augment it by the introduction of additional aggregated concepts useful as "building blocks" for programming.

By making the general purpose language highly standardized and available on many types of computers, the application languages also become easily transferable, and at the same time the software development costs for the computer manufacturers may be retarded from the present rapid increase.

1.3 The basic characteristics of SIMULA 67

1.3.1 Algorithmic capability

SIMULA 67 contains most features of the general algorithmic language ALGOL 60 as a subset. The reason for choosing ALGOL 60 as starting point was that its basic structure lent itself to extension. It was felt that it would be impractical for the users to base SIMULA 67 on yet another new algorithmic language, and ALGOL 60 already had a user basis, mainly in Europe.

1.3.2. Decomposition

In dealing with problems and systems containing a large number of details, decomposition is of prime importance. The human mind must concentrate; it is a requirement for precise and coherent thinking that the number of concepts involved is small. By decomposing a large problem, one can obtain component problems of manageable size to be dealt with one at a time, and each containing a limited number of details. Suitable decomposition is an absolute requirement if more than one person takes part in the analysis and programming.

The fundamental mechanism for decomposition in ALGOL 60 is the block concept. As far as local quantities are concerned, a block is completely independent of the rest of the program. The locality principle ensures that any reference to a local quantity is correctly interpreted regardless of the environment of the block.

The block concept corresponds to the intuitive notion of "sub-problem" or "sub-algorithm" which is a useful unit of decomposition in orthodox application areas.

A block is a formal description, or "pattern", of an aggregated data structure and associated algorithms and actions. When a block is executed, a dynamic "instance" of the block is generated. In a computer, a block instance may take the form of a memory area containing the necessary dynamic block information and including space for holding the contents of variables local to the block.

A block instance can be thought of as a textual copy of its formal description, in which local variables identify pieces of memory allocated to the block instance. Any inner block of a block instance is still a "pattern", in which occurrences of non-local identifiers, however, identify items local to textually enclosing block instances. Such "bindings" of identifiers non-local to an inner block remain valid for any subsequent dynamic instance of that inner block.

The notion of block instances leads to the possibility of generating several instances of a given block which may co-exist and interact, such as, for example, instances of a recursive procedure. This further leads to the concept of a block as a "class" of "objects", each being a dynamic instance of the block, and therefore conforming to the same pattern.

An extended block concept is introduced through a "class" declaration and associated interaction mechanism such as "object references" (pointers), "remote accessing", "quasi-parallel" operation, and block "concatenation".

Whereas ALGOL 60 program execution consists of a sequence of dynamically nested block instances, block instances in SIMULA 67 may form arbitrary list structures. The interaction mechanisms which are introduced, serve to increase the power of the block concept as a means for decomposition and classification.

1.3.3 Classes

A central new concept in SIMULA 67 is the "object". An object is a self-contained program (block instance), having its own local data and actions defined by a "class declaration". The class declaration defines a program (data and action) pattern, and objects conforming to that pattern are said to "belong to the same class".

If no actions are specified in the class declaration, a class of pure data structures is defined.

Example

```
class order (number); integer number;
   begin integer number of units, arrival date;
         real processing time;
   end;
```

A new object belonging to the class "order" is generated by an expression such as

"new order (103)"

and as many "orders" may be introduced as desired.

The need for manipulating objects and relating objects to each other makes it necessary to introduce list processing facilities (as described below).

A class may be used as "prefix" to another class declaration, thereby building the properties defined by the prefix into the objects defined by the new class declaration.

Examples:

```
order class batch order;
   begin integer batch size;
         real setup time;
   end;

order class single order;
   begin real setup time, finishing time,weight; end;

single order class plate;
   begin real length, width; end;
```

New objects belonging to the "sub-classes" - "batch order" "single order" and "plate" all have the data defined for "order", plus the additional data defined in the various class declarations. Objects belonging to the class "plate" will, for example, comprise the following pieces of information: "number", "number of units", "arrival date", "processing time", "setup time", "finishing time", "weight", "length" and "width".

If actions are defined in a class declaration, actions conforming to this pattern may be executed by all objects belonging to that class. The actions belonging to one object may all be executed in sequence, as for a procedure. But these actions may also be executed as a series of separate subsequences, or "active phases". Between two active phases of a given object, any number of active phases of other objects may occur.

SIMULA 67 contains basic features necessary for organizing the total program execution as a sequence of active phases belonging to objects. These basic features may be the foundation for aggregated sequencing principles, of which the class SIMULATION is an example.

1.3.4 Application language capability

SIMULA 67 may be oriented towards a special application area by defining a suitable class containing the necessary problem-oriented concepts. This class can then be used as prefix to the program by the user interested in this problem area.

The unsophisticated user may restrict himself to using the aggregated, problem-oriented and familiar concepts as constituent "building blocks" in his programming. He may not need to know the full SIMULA 67 language, whereas the experienced programmer at the same time has the general language available, and he may extend the "application language" by new concepts defined by himself.

As an example, in discrete event system simulation, the concept of "simulated system time" is commonly used. SIMULA 67 is turned into a simulation language by providing the class "SIMULATION" as a part of the language,

(in this case provided with the compilers).
In the class declaration

```
class SIMULATION;
   begin ..... end;
```

a "time axis" is defined, as well as two-way lists (which may serve as queues), and also the class "process" which gives an object the property of having its active phases organized through the "time axis".

A user wanting to write a simulation program starts his program by

```
SIMULATION begin .....
```

in order to make all the simulation capabilities available in his program. If he himself wants to generate a special-purpose simulation language to be used in job-shop analysis, he may write:

```
SIMULATION class JOBSHOP;
   begin ..... end;
```

and between "begin" and "end" define the building blocks he needs, such as

```
process class crane;
   begin ..... end;

process class machine;
   begin procedure datacollection; .....
   .....
   end;
etc.
```

The programmer now compiles this class, and whenever he or his colleagues want to use SIMULA 67 for jobshop simulation, they may write in their program

```
JOBSHOP begin .................
```

thereby making available the concepts of both "SIMULATION" and "JOBSHOP".

This facility requires that a mechanism for the incorporation of separately compiled classes is available in the compiler (see section 15).

1.3.5 List processing capability

When many objects belonging to various classes do co-exist as parts of the same total program, it is necessary to be able to assign names to individual objects, and also to relate objects to each other, e.g. through binary trees and various other types of list structures. A system class, "SIMSET", introducing circular two-way lists is a part of the language.

Hence basic new types, "references", are introduced. References are "qualified", which implies that a given reference only may refer to objects belonging to the class mentioned in the qualification (or belonging to subclasses of the qualifying class).

Example:

```
ref(order)next, previous;
```

The operation of making a reference denote a specified object is written ":-" and read "denotes".

Example:

```
next :- new order (101); previous :- next;
```

or (also valid since "plate" is a subclass of "order")

```
next :- new plate(50);
```

Data belonging to other objects may be referred to and used by "remote accessing", utilizing a special "dot notation".

Example:

```
if next.number ≥ previous.number then .....;
```

comparing the "number" of the "order" named "next" with the "number"of the "order" named "previous".

The "dot notation" gives access to individual pieces of information. "Group access" is achieved through "connection statements".

Example:

```
inspect next when plate do begin ..... end;
```

In the statement between begin and end all pieces of information contained in the "plate" referenced by "next" may be referred to directly.

1.3.6 String handling

SIMULA 67 contains the new basic type "character". The representation of characters is implementation defined.

In order to provide the desired flexibility in string handling, a compound type called "text" is introduced. The "text" concept is closely associated with input/output facilities.

1.3.7 Input/output

ALGOL 60 has been seriously affected by the lack of standardized input/output and string handling. Clearly a general purpose programming language should have great flexibility in these areas. Consequently, input/output are defined and made a standardized part of SIMULA 67.

1.4 Standardization

For a general purpose programming language it is of paramount importance that while the language is uniquely defined and at the same time under strict control, it may be extended in the future.

This is achieved by the SIMULA Standard Group, consisting of representatives for firms and organizations having responsibility for SIMULA 67 compilers. The statutes lay down rigid rules to provide for both standardization and future extensions.

The SIMULA definition which is required to be a part of any SIMULA 67 system is named the "SIMULA 67 Common Base Definition".

1.5 Language definition

The language definition given in the following sections must be supplemented by the formal definition of ALGOL 60 [1]. The syntactic definitions given in this report are to be understood in the following way.

1) Syntactic classes referred to, but not defined in this report, refer to syntactic definitions given in [1].

2) Definitions in this report of syntactic classes defined in [1] replace the corresponding definitions given in [1].

3) Any construction of the form

   <ALGOL some syntactic class>

   stands for the list of alternative direct productions of <some syntactic class> according to the definition given in [1].

4) The comment conventions given in [1] is extended in that the convention for "end-comment" is replaced by:

   {end <any sequence not containing ;, end, else, when or otherwise>} → {end}