Portable Simula Revisited

Programmer's Reference

Working Draft

(as of xx februar 2019)

Table of Contents

1. INTRODUCTION

This is a new Simula System created by the Open Source Project 'Portable Simula Revisited'.
The project was initiated as a response to the lecture held by James Gosling at the 50th anniversary of Simula in Oslo on 27th September, 2017. He mentioned that most programming Languages have been re-implemented in Java but not Simula.

Therefore this Portable Simula System is written in Java and compiles to Java code with one exception; the Goto Statement need to be corrected in the byte code, which is done automatically.

Simula is considered the first object oriented programming language. It was originally designed for solving dicrete event simulations as an extension to the Algol Language. But it quickly turned out to be a general purpose programming language. Among the main advantages of Simula are the introduction of the **'class'** declaration which lead to very cost-effective program development, and The standardisation of the language which means that Simula programs are easily transferred Between different Simula Systems.

One of the most important aspects of Simula is the possibility to define and compile a set of tools oriented towards some specific application. Three important examples of this possibility are included as standard facilities: The standard input-output package (chapter 10), list processing (chapter 11) and the package for discrete event simulation (chapter 12).

This Portable Simula System handles the full Simula language as defined by the Simula Standard adopted by the Simula Standards Group, except as noted in the last section of this chapter ("Deviations from Standard SIMULA").

THE PORTABLE SIMULA SYSTEM

The Portable SIMULA system is composed of three main entities: the Simula Editor, the Language Compiler, and the Execution Environment.

The program source is composed as described in the Simula Standard. It consists of statements which represent logical steps of a problem solution, and declarations which specify the representation of data operated upon by these statements.

The SIMULA Compiler

The compiler analyses the program and generates an equivalent Java program, containing instructions corresponding to the SIMULA statements. It also assigns storage for constants, creates references to external programs if any, and prepares for the dynamic generation of the storage locations for objects corresponding to the declarations of the program.

TARGETSYSTEM SIMULA compiler is a four-phase compiler. A large part of the compiler is actually programmed in SIMULA itself. It is described in broad terms in section 17.1.

The compiler operates in the normal manner, reading the source program from a series of images presented in a runstream or, more commonly, saved in a program file. The main output of the compilation consists of a relocatable binary program, just like any other TARGETSYSTEM language processor. In addition the compilation of a separately compiled module leads to output of so-called attribute elements. This is described in more detail in chapter 6.

The SIMULA Execution Environment

The SIMULA execution environment consists of the system software (such as e.g. OPSYSINFO, etc), the SIMULA Run Time System (RTS), and the Environment Interface. In addition all hardware employed by these components and by the program belong to this environment.

The Run Time System and the Environment Interface are described in chapters 17 and 18.

An important component of the execution environment is the SIMULA symbolic debugger "SIMOB"; it is described in chapter 21.

THE NOTATION

The notation used to specify the syntax of the SIMULA language is based on the Backus-Naur Form. The exact definition of the notation is found in the beginning of chapter 1. Although this notation may seem somewhat complicated at first glance, it provides for an exact definition of the composition of SIMULA programs, and is superior in this respect to more informal description methods.

The syntactic specification of the language is given as a set of rules, each defining a specific language construct. The general format of such a rule is

```
   left-side
      =  first-alternative
      |  second-alternative
      |  ... etc.
```

where the "left-side" is a so-called meta symbol that is an identifier which represents some language construct; the meta symbol will hint at the actual nature of the construct. A right-hand side alternative may be either a meta symbol, a terminal symbol such as _integer_ or 1066, or it may consist of a sequence of meta symbols and/or terminal symbols (a "terminal symbol" is a string of characters that may actually be part of the program).

Such a sequence of terminal and meta symbols in a language rule implies the concatenation of the text that they ultimately represent. Within chapter 1 the concatenation is direct; no characters may intervene. In all other parts of this manual the concatenation is in accordance with the rules set out in chapter 1.

Example   The production

```
   identifier
      =  letter  { letter | digit | _ _ }
```

describes a legal identifier as consisting of at least one letter, possibly followed by a sequence of letters, digits, and underlines. Since this rule occurs in chapter 1, it follows that no blank can occur within a legal SIMULA identifier (the reason why blanks occur anyway in the rule is to make it readable). All identifiers occurring within program fragments given in this manual are legal identifiers, whereas for instance "2ab" and "SIM$*RTS$" are illegal identifiers: 2ab starts with a digit and SIM$*RTS$ contains the characters '$' and '*' which may not occur within an identifier ('*' is the multiplication operator, '$' may occur only within text constants and in comments and directives).

Deviations from '*Simula Standard*'

The current version of this SIMULA system differs from the *Simula Standard* in some areas. Following the chapters of *Simula Standard*, this is a complete list:

```
5 Virtual labels and switches are not part of this
  implementation.

6 External non-SIMULA procedures are NOT IMPLEMENTED.

9 The name specified parameter to the random drawing procedures
  is not used.Instead, the default seed in the Java Library is
  used. This is not a real deviation since Simula Standard says
  BasicDrawing is implementation defined.

  The <real-type> parameter to the 'histd' procedure must be
  type real.

10 The File procedure setaccess is not fully implemented
```

Implementation Defined Aspects

Within the Simula Standard, certain aspects ot the language are left open to be decided by the actual implementation. Section 6. gives a complete list of these, together with a specification of the decision taken om each subject for this Portable Simula System.

Information on how to install the system and about system limitations etc. is found in section 7.

About this Manual

This manual has repeated references to the Simula Standard. Note that the Standard was not intended as a text book on Simula, therefore it is rather formal in style in order to be as precise as possible. We refer to one of the text books on the marked, for instance "An Introduction to programming in Simula" by Rob Pooley.

The remainder of the manual contains implementation dependent information of a practical nature, such as e.g. how to compile and execute a source program using the Simula Editor.

In addition information is given on system limitations, file search strategies etc.

2. Download and Installing the System

To download and install this Simula System you may visit the web-page:

https://portablesimula.github.io/github.io/

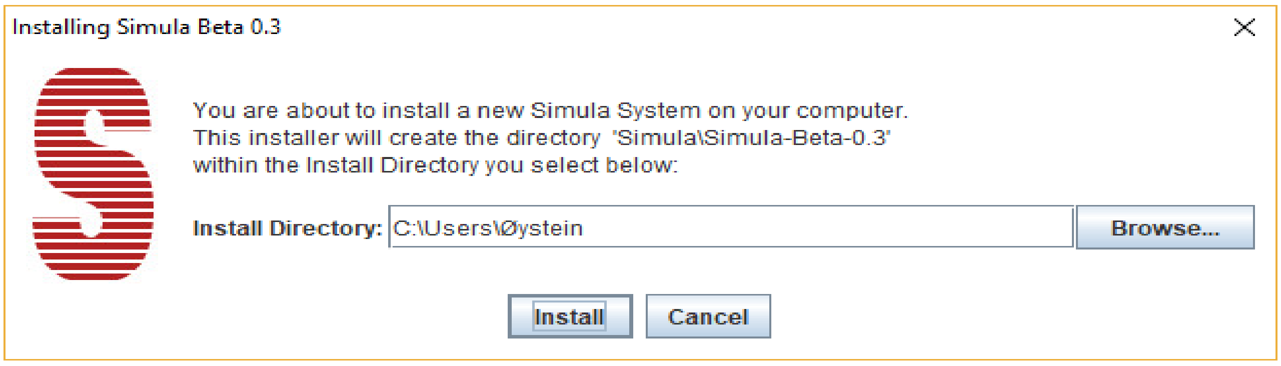
Select 'Simula Download' and follow the instructions.

Just when your computer is about to download the setup file it probably give a warning message. This message states that you are downloading from a unsafe source. You must ignore this message.

When the 'setup.jar' is completely downloaded its time for executing it. This is done by clicking on it on Windows PC's.

< add description for Unix, MAC, Linux … >

When setup.jar is running you are first asked for a install directory:



Installing Simula Beta 0.3

You are about to install a new Simula System on your computer.
This installer will create the directory 'Simula\Simula-Beta-0.3'
within the Install Directory you select below:

Install Directory: C:\Users\Øystein

In this case the Simula System will be installed in the following directory structure.

C:\Users\Øystein

\Simula	(This Directory is called 'Simula Home')
\Simula-Beta-03	(This Directory is called 'Release Home')
\rts …	(Simula Runtime System)
\samples …	(Sample Simula Programs)
simula.jar	(Executable Simula Compiler)
ReleaseNotes.txt	

3. Activating the Simula Compier

The Simula Compiler is normally activated through a command-line of this form:

```
java  -jar  release-home\simula.jar
```

In this simple case the Simula Editor (see the following pages) is started.

On Windows an activation file RunSimulaEditor.bat is created during setup and placed in the Release Home directory together with the 'simula.jar' file. On Unix, Linux, MAC etc. a corresponding file RunSimulaEditor.sh is genereated.

You can move these script files wherever you want, for example, to the desktop. Alernatively, on Windows you may create a link to 'simula.jar' and put it on the desktop. Icons are available in the \icon sub-directory if you want to decorate such 'buttons'.

General Case

In special rare situations you may use the general version of the command-line form:

```
java   [java-options]  -jar  release-home\simula.jar  [ simula-sourceFile ]  [simula-options]
```

Java-options
: are described in the relevant Java Technical Dokumentation.

Simula-sourceFile
: is the file containing the Simula text to be compiled and runed.

possible simula-options include:

```
-help                     Print this synopsis of standard options
-noexec                   Don't execute generated .jar file
-nowarn                   Generate no warnings
-verbose                  Output messages about what the compiler is doing
-keepJava <directory>     Specify where to place generated .java files
                          Default: Temp directory which is deleted upon exit
-output <directory>       Specify where to place generated executable .jar file
                          Default: Current workspace\bin
```

4. File Handling Conventions

5. Additional Facilities

5.3 Directive Lines

The following compiler directives are defined Acording to the rules of Simula Standard sect.1.1:

If the first character of a source line is "%" (percent) the line as a whole is a directive line.
A directive line serves to communicate information to the compiler and consequently its meaning is entirely implementation-dependent, with the following single exception. If the second character is a space, the line has no significance; it may be used for annotation purposes. .

%BOUNDCHECK ON / OFF

> Controls generation of array bound checking, if possible on the given implementation.
>
> This S-PORT directive is ignored by this implementation.

%COPY file-name

> Will cause the compiler to include the indicated file at this place in the source input file, as if the text was actually written in the module containing %COPY. COPY (or INSERT) may occure in the included file. The included lines are always counted, i.e. they wil be numbered sequenctially starting from the number of the line of the COPY-directive.

%DEFINITION

> This directive is relevant only in a separately compiled procedure or class. The compiler will regard this compilation as a prototype definition only, i.e. the module is checked for syntactic and semantic correctness, and the appropriate attribute element is produced, but no code generation is performed.
>
> This S-PORT directive is ignored by this implementation.

%EOF

> When the compiler recognizes this directive it reacts as if the end of the source file was encountered, i.e. it terminates reading and, if this module was inserted (see INSERT / COPY), source scanning will continue after the INSERT (COPY) directive in the enclosing file.
>
> This S-PORT directive is ignored by this implementation.

%INSERT file-name

> Will cause the compiler to include the indicated file at this place in the source input file. INSERT may occur in the included file. In constrast to COPY, the included lines are not Counted (they will all be numbered with the line number of the line containing the outermost INSERT). Furthermore, if the source is being listed, listin is turned off before the inclusion and turned on again when reading continues after this directive.
>
> This directive is treated as %COPY in this implementation.

%LIST ON / OFF
%NODUMMY ON / OFF
%NONECHECK ON / OFF
%NOSOURCE
%PAGE
%QUACHECK ON / OFF
%REUSE ON / OFF

These S-PORT directives are ignored by this implementation.

%SELECT string

Set or reset selectors for conditional compilation, see next section below.

%SETOPT string

Set or reset options and trace switches, see next section below.

%SETSWITCH name-string value-string

Will set compiler switch "name" to the value "value". The facility is intended for compiler maintenance, and is not explained further.

%SLENGTH last-column
%SLENGTH last-column
%SPORT ON / OFF
%TITLE page-heading

These S-PORT directives are ignored by this implementation.

%+string / %-string

Conditional compilation control, see next section below.

5.4 Conditional Compilation

This implementation makes it possible, on a line by line basis, to conditionally include or exclude source text from a compilation.

During input of the source text, the compiler maintains a set of boolean variables or selectors, corresponding to the letters of the (English) alphabet. Initially, these selectors are all reset. They may be set (reset) either by means of SELECT directives embedded in the source text or before compilation is activated by using compiler options.

Note: The case of the selector letters is significant.

If a source line contains "%+" or "%-" as its first two characters, the line is a conditional line. Such a line will be compiled conditionally, depending on the value ot the selector(s) following immediately after. If no selector follows the line is treated as an illegal directive and ignored.

The SELECT Directive

The directive takes the form:

```
%SELECT  character-sequence
```

First, all selectors are reset. Then, for each character in the sequence, the corresponding selector is set.

Conditional Lines

A conditional line takes the form:

```
%selector-expression  ….
```

where " …. " represents the line to be conditionally included
and the selector-expression has the form:

```
Selector-expression
        =  selector-group  {  selector-group  }

Selector-group
        =  +  letter  {  letter  }
        |  -   letter  {  letter  }
```

i.e. a string of letters and signs, with the first character being a sign. The selector-expression is terminated by a SP.

Within each group, the values of the named selectors are tested. If the group I positive (prefixed By '+') and all tested selectors are set, or in the case of a negative group all tested selectors are reset, the group is deleted from the line, and the process is repeated for the next group (if any). If all selector groups have been deleted in this manner, the initial '%' and the terminating space is likewise deleted, and the resulting is treated as if it was read from source.

Note: The resulting line may well be a directive line.

If, on the other hand, a positive (negative) group contains a reset (set) selector, the line as whole is skipped.

Example:

%+MD-f x:=x-1;	the statement is compiled only when selectors 'M' and 'D' are set and 'f' is reset.

6 Structure of the SIMULA System

6.1 The SIMULA Compiler

6.2 The Runtime System

7. Implementation Defined Aspects