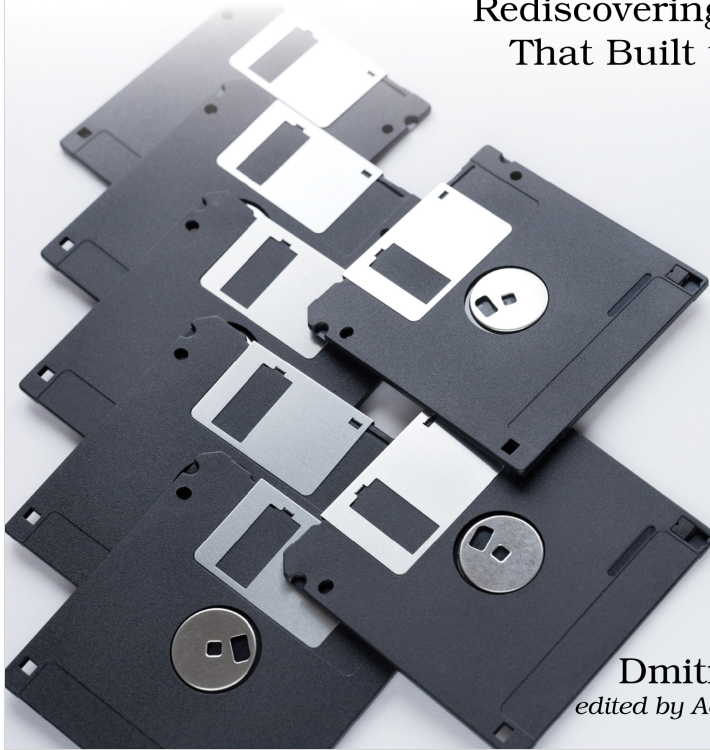


The
Pragmatic
Programmers

Seven Obscure Languages in Seven Weeks

Rediscovering the Tools
That Built the Future



Dmitry Zinoviev
edited by Adaobi Obi Tulton

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit <https://www.pragprog.com>.

Copyright © The Pragmatic Programmers, LLC.

This chapter offers you a gift. Instead of one obscure language, it gives you two: Simula and ALGOL, because Simula is ALGOL in disguise (specifically, Simula-67 is a superset of ALGOL-60). Beyond OOP, Simula's origins in simulation will introduce you to coroutines and discrete event modeling, enabling you to simulate complex systems—a skill increasingly crucial in data-driven decision-making.

But don't get too excited yet; ALGOL had a massive influence on computing in general and specifically on programming languages, but, forgotten for any practical purpose, it's remarkably boring.

ALGOL is the second oldest programming language, designed in 1958 as a direct successor to Fortran. Its antiquated features are a clear reminder of its roots in the early days of computing. ALGOL is characterized by its heavy syntax, reliance on GOTO statements, a clear divide between functions and procedures, lack of dynamic memory allocation, and compatibility with punch cards rather than files.

Such shortcomings, among others, must have prompted the emergence of Simula between 1962 and 1967. Simula stepped in to supplant ALGOL, absorbing its core components while enhancing its capabilities. Developed initially for CDC 3300, it was soon ported to CD 3600, CD 6600, UNIVAC 1108, IBM 360/370, and other computer systems.

Simula rightly prides itself as the pioneer of object-oriented programming. Though the individual components of this paradigm had been present since 1960, Simula was the first to implement the framework fully. Shortly after, in 1972, Smalltalk followed suit, reinforcing the object-oriented programming approach.

"It's ALGOL!"

Boring or not, ALGOL is Simula, and you cannot learn one without learning the other. Let's start with the traditional "Hello, world!" chant to illustrate using comments and compound statements.

```
simula/hello.sim
% A comment
! Another comment ;
BEGIN
    OutText("Hello World!");
    COMMENT One more comment ; OutImage;
END of Program
```

An unterminated comment begins with % and extends to the end of the line. Terminated comments begin with ! or COMMENT and extend to the first semicolon

because both are statements, and statements must always conclude with a semicolon. Everything after the END clause to the end of the line is ignored too.

A *compound statement* is a sequence of single or other compound statements wrapped in BEGIN and END clauses. It can be utilized in any situation where a single statement would be expected. A Simula program is typically one compound statement.

Compound Statements



Simula compound statements correspond to the C/C++/Java compound statements enclosed in curly braces {} or Python blocks indented by the same amount. Compound statements were first introduced in ALGOL.

ALGOL-60 and Simula are case-insensitive languages. As a convention, consider typing keywords in uppercase letters and identifiers in mixed-case letters for better readability and consistency.

Since Simula is a superset of ALGOL-60, treat any reference to Simula as an implicit reference to ALGOL, and the other way around, until you reach the section [“It’s Simula!”, on page ?](#). The histories of these two languages diverge after that point.

Glancing at Variables, Data Types, and Operators

A variable must be declared at the beginning of the statement in which it’s used before any executable statement (such executable statements are called *sentences*). A declaration (or a *qualification*, as it’s known in the original Simula documentation) includes the name and the data type.

The set of primitive data types in Simula is almost standard for its time. It has INTEGER numbers (26, -12), REAL numbers (3.14159, 2E-8), BOOLEAN constants (TRUE and FALSE), and EBCDIC CHARACTER (*). Explicit type conversion is not supported.

Operators in Simula generally behave as you would expect. Many operators come in two forms: “classical” ALGOL-60 (with non-EBCDIC characters) and “modern” (only with EBCDIC/ASCII characters). The arithmetic operations include addition (+), subtraction (-), multiplication (× or *), division (/), integer division (÷ or //), and exponentiation (↑ or **).

Logical operators encompass the following: greater than (>), less than (<), less than or equal to (≤ or <=), greater than or equal to (≥ or >=), inequality (≠, ≠, or <>, which compares values), reference inequality (≠=, which compares refer-

ences), equality (=, compares values), and reference equality (==, compares references).

The language also includes the following relational operators: negation (\neg or NOT), conjunction (\wedge or AND), disjunction (\vee or OR), implication (\supset or IMP, interpreted as “B follows from A” or “NOT A OR B”), and equivalence (\equiv or EQV, which is the same as NXOR).

Comparison



Operators = and <> correspond to Python language operators == and !=. Operators == and != correspond to Python language operators is and is not.

Characters

Simula characters are enclosed in single quotes. A few procedures are provided to convert them to and from integer character codes and to test their associations with character classes.

- rank(c). Returns the integer character code (compare ord(c) in Python).
- char(n). Returns the character that corresponds to the code n (compare chr(n) in Python).
- digit(c). Returns TRUE if c is a decimal digit.
- letter(c). Returns TRUE if c is a letter (according to the local definition of letters).

Text Objects and Operations

Simula offers two additional data types specific to its structure: REF and TEXT. The REF type is a reference to an object, denoted as NONE if no object is referenced. The TEXT type is used for text strings—for example, "Hello, world!". When not initialized, its value is NOTEXT.

A Simula reference is an alternate identifier for an existing object. An instance of the TEXT type is a reference to a compound object—a *text descriptor*.

Unlike the languages we remember, Simula treats text not as a character array but as a memory file. The text descriptor contains information about the text area (text buffer), including its memory address, size, and current position within the text (it starts at 1). Typically, this position points to the first uninitialized character, but you can reposition it as desired. If T is a text, then Integer T.Pos returns its current position, T.SetPos(n) sets the current position

to n , Integer `T.Length` returns the text area size, and Boolean `T.More` checks if the position is at the end of the text area.

In Simula, you can create a text object using two methods:

- Allocate a blank space-initialized text area of size n with the procedure `Blanks(n)` and then initialize by assigning a literal string.
- Copy a literal string s with the procedure `Copy(s)`.

Note that Simula has two assignment operators. `:=` is used for value assignment, and `:-` is used for reference assignment:

```
TEXT t1, t2;
t1 :- Blanks(10);      ! Reference assignment ;
t1 := "Hello,";        ! Value assignment ;
t2 :- Copy("World!");  ! Reference assignment ;
```

Several procedures can be utilized to manipulate texts in Simula, including the following:

- `T.GetChar`. Returns the current character and advances the current position.
- `T.PutChar(c)`. Inserts the character c at the current position and advances the current position.
- `T.Sub(p,n)`. Creates a subtext of length n , starting at the position p .
- `T.Strip`. Eliminates white spaces (blanks) at the end of text area.
- `T.GetInt`. Interprets the text as an integer number.

- T.GetReal. Interprets the text as a real number.
- T.PutInt(n). Appends an integer number n.
- T.PutFix(n,w). Appends an integer number n of a fixed width w.
- T.PutReal(n,w). Appends a real number n of a fixed width w.

As an example, the following code fragment greets the author. It allocates a 64-character text area, copies the greeting, appends the author's initials, removes unused characters, and displays the results.

```
simula/greet.sim
BEGIN
  TEXT txt;
  txt := Blanks(64);
  txt := "Hello, ";
  txt.SetPos(8); ! Skip over the greeting! ;
  txt.PutChar('D');
  txt.PutChar('Z');
  txt.Strip; ! Remove the trailing blanks ;
  OutText(txt);
  OutText("!");
END;
```

If cim is installed on your system, you can compile and run the program as follows:

```
/home/dzseven> cim greet.sim
Compiling greet.sim:
gcc -g -O2 -c greet.c
gcc -g -O2 -o greet greet.o -L/usr/local/lib -lcim
/home/dzseven> greet
Hello, DZ!
```

Arrays

Simula arrays are static and optionally multidimensional. The type, size, index ranges, and number of dimensions of an array must be declared at compile time. Indices can start from any value, not limited to 0 or 1. Array elements can be accessed and modified using parentheses notation.

```
INTEGER ARRAY age, weight(0:16); ! A vector of length 17 ;
CHARACTER ARRAY chess(1:8,1:8); ! A 2D array 8x8 ;
age(0) := 16;
chess(1,1) := 'Q';
```

Simula performs a runtime index check and will terminate your program if an index is out of bounds.

Arrays



Simula arrays are comparable to statically declared arrays in C and fixed-size arrays in Java.