



## **Semiotic Modeling for De-Duplication of Text Strings Using A Word-Stack Approach**

J.B. Phillips, R. Wiltshire, P. Prueitt, G. Price

Engineering Development Institute &  
Highland Technologies

### Introduction

One of the most fundamental relationships in an algebraic structure is the property of equality. The reflexive property of equality (Selby and Sweet, 1969) states that

$$(1) \quad A = A$$

and the commutative (otherwise referred to as the symmetric) property of equality (Kuratowski and Mostowski, 1968) states that if

$$(2) \quad A(\text{operator})B = C$$

then

$$(3) \quad B(\text{operator})A = C$$

However, these properties assume an altogether different meaning if A or B can change over time. Thus, if

$$(4) \quad A_{|t=0} \neq A_{|t=t1}$$

then the entire concept of equality is eroded substantially.

Similarly, the concept of a function hinges on the constancy of values. If

$$(5) \quad f(A_{|t=0}) \neq f(A_{|t=t1}),$$

which is part of the definition of a function, contingent upon Eq. (4), then the concept of a function also assumes a dramatically different meaning. Moreover, assume that

$$(6) \quad A_{|t=t2} \neq A_{|t=t3}$$

Then if

$$(7) \quad f(A_{|t=t2}) \neq f(A_{|t=t3}),$$

it can be concluded that f is not a true function.

Consider the following definition of a function (Michel and Herget, 1981):

Let X and Y be non-void sets. A function f from X into Y is a subset of  $X \times Y = \{(x, y): x \in X, y \in Y\} \ni \forall x \in X \exists$  a unique  $y \in Y \ni (x, y) \in f$ . The set X is called the domain of f, and we say that f is defined on X. The set  $\{y \in Y: (x, y) \in f \text{ for some } x \in X\}$  is called the range of f, and is denoted by  $\mathfrak{R}(f)$ .  $\forall (x, y) \in f$ , we call y the value of f at x and denote it by f(x). We write  $f:X \rightarrow Y$  to denote the function f from X into Y, or mapping X into Y.

It is noteworthy that, according to the above definition, a function is a set of ordered pairs, with the first member being an element of set X, and

the second member being an element of set Y. X and Y can be any sets, so if we state that X is a set of documents, and Y is a set of text files (or equivalently, a set of binary numbers, admittedly quite long binary numbers, encoding text files), then we may state that scanning a document into a text file is a function if and only if every time a document is scanned, the same text file is produced (Gilbert, 1970).

Clearly, we want scanning to be a function, since constancy of the scanning process is desired. Unfortunately, this goal is complicated by a need for representations of documents in a database to evolve as additional documents are compiled into said database (Bouchaffra and Meunier, 1995; Rijsbergen, 1979; Robertson and Harding, 1984). If a keyword system (or equivalent indexing system) is used to represent the content of text strings in a database, then obviously a small database will require fewer keywords to match document content than a large database will. Neural networks (Belew, 1989; William, 1990) and a Markovian Random Field Machine (Bouchaffra and Meunier, 1995) have been proposed as solutions to this portion of the problem. Both methodologies have displayed promise. Regardless of the methodology, some system comparing UNits of INformation (otherwise referred to as UNIFs; Meunier, et al., 1993; Meunier, et al., 1987) between new additions to a database and existing database members needs to be implemented.

How do these concepts relate to semiotic modeling? The scientific method tells us that if a process (or function, in the case at hand) is carried out consistently on an identical input, the results also should be identical. Unfortunately, when scanning text strings for input into a text file, this is not always the case. Some examples are:

- \* Punctuation: The small size of many punctuation marks can result in an inconsistent scanning of punctuation.

- \* Misread letters: Depending on the font, an upper case “A” (for example) can sometimes be scanned as “Fl” (upper case F, lower case ell).

- \* Spaces: OCR software performs with a great degree of variation when inserting spaces between words. For example, there is less space between M and N than there is between A and V (MN vs. AV, by inspection). The variations in the spaces required between different pairs of letters can result in an extra space inserted in the middle of a word, or (more commonly) two or more spaces inserted between adjacent words, when only a single space is intended.

Therefore, when a hard copy of text is scanned into a text file, it is not only conceivable, but even likely, that the results of the scan will not be the same when conducted twice in a row. (Even such factors as variation in the position (including angle) of the hard copy being scanned can have a significant effect in this regard.) This leads to an interesting question: Is the “text” actually the hard copy, or is it the file? In other words, is the file a representation of the “real” object (specifically, the hard copy), or is the hard copy a representation of the “real” object (specifically, the file)? Ultimately, it must be recognized that the text’s author’s thoughts and words are the true object (albeit one can argue that even the author’s words are only a representation of the thoughts, i.e., the “true” object), and both the hard copy and the file are attempts to communicate those thoughts to other individuals (or to machines). Herein lie the roots of artificial intelligence and machine-based understanding of human thoughts (Partridge and Wilks, 1990). Fortunately, computers have been constructed to function in a manner similar to the human mind, and just as the human mind is innately a symbolic system that constructs symbols, manipulates them, and relates them to real objects external to the mind, so do computers receive inputs of symbols, manipulate them, and provide output of symbols (Meunier, 1998). For the purposes of the discussion at hand, this question does not need to be resolved immediately, but it remains of interest, and has implications for future work.

What can be concluded about the lack of constancy in results from a document scanning procedure? It is patently obvious that the hard copy does not undergo meaningful change from one scan to the next (although its position on the scanner can vary). However, one would be hard pressed to state that the scanning procedure is not a true function, in the mathematical sense of the word. (In an ideal scanning process, scans of identical input should yield identical output.) Natural scatter of output data is acceptable, at least to a certain degree; moreover, natural scatter is not inconsistent with the concept of a mathematical function. The problem is made more complex by the fact that one goal of a de-duplication procedure is to scan different hard copies and determine which ones contain text identical to ones previously scanned.

Within this context, the following problem has been presented to the scientific community: How can hard copies of classified material (much of which is cable traffic), which can be years or decades old, be compared with text previously scanned into a database? This problem stems from the Freedom of Information Act and recent executive orders to declassify any material not currently considered to be vital to national security. (Although this

work was conducted with an eye toward the American government, and text in English, the concepts are universally applicable.)

Declassifying the millions of documents that apply to the situation at hand is expected to be a labor-intensive process. A significant step in reducing the labor requirements is to determine which documents (or text strings within documents) are copies of each other. The project will entail scanning all of the documents, discarding copies, and processing the remaining documents for a decision on declassification.

Therefore, one of the earliest crucial steps in declassifying the documents is determining which hard copies are duplicates of each other. The subject of information retrieval has been examined in some detail with respect to applications similar to these (Bouchaffra and Meunier, 1995). It has been common for multiple hard copies of cable traffic to be stored in divergent locations, often for decades. Cable traffic poses some unique challenges with respect to the problem of de-duplication. First, all of the characters (at least in most cable traffic) are upper case letters. There is no punctuation, and no lower case characters comprise the text. However, in contrast to modern word processors, there is no adjustment for spacing between words, so scanning superfluous spaces is a problem requiring special attention. As an example of the appearance of cable traffic, what normally would be written:

‘The quick, brown fox jumped over the lazy dog.’

which becomes, in cable traffic

THE QUICK COMMA BROWN FOX JUMPED OVER THE LAZY DOG STOP

The text generally is typed in standard, block letters, with no punctuation, or with punctuation spelled out in text (i.e., “COMMA” denoting a comma and “STOP” denoting the end of a sentence in lieu of a period, *vide supra*). Therefore, the major remaining roadblock (of the three presented previously, *vide supra*) is the problem of spacing.

## Approach

Although there are a number of means of approaching the problem of variable spacing, each has some inherent disadvantages associated with it. For example, one simply could process the text string as a long string of characters, with a function or procedure that deletes spaces whenever they are encountered. However, then the text string would exist as a variable consisting of a single long character string, and long documents would require large amounts of memory to store the variable. Moreover, manipulating a variable of this nature would be cumbersome.

In contrast, a word stack methodology involves storing each word as a character string variable. Moreover, the spaces between words are not stored, because the presence of a space in the input simply signifies that a word has ended, and that the next variable begins with the next occurrence of a (non-space) character. It does not matter how many spaces exist, the spaces do not get stored in the word stack. This approach is ideally suited to the problem at hand.

## Experimental

C++ version 4.52 (courtesy Borland) was chosen as the programming language. The advent of object-oriented programming languages such as C++ (Sedgewick, 1992), Visual Basic (Aitken, 1996), and Java (Walsh, 1996) has revolutionized scientific programming, and the advantages of object-oriented methodologies have become widely recognized and accepted (Rumbaugh, et al., 1991).

Within that context, an object-oriented methodology was implemented (Sodhi and Sodhi, 1996). One part of an object-oriented approach consists of using classes to protect data, and controlling access to and manipulation of the data through the use of class member functions. The concept of a class is rooted in set theory (Muller, 1976), and lends itself well to be adapted to problems such as the one at hand.

The code that was used to implement the word stack is contained in the Appendix [see the online version of *AS/SA* to download the executable].

The word stack class code was adapted from string classes developed previously (McMonnies and McSporrán, 1995; Capper, 1994; Prata, 1995). It has been tailored to compare text strings on a word-by-word basis, while ignoring spaces between words (*vide infra*, Discussion Section). If the number of words in each of the two inputs is the same, and each word in each input is the same as the corresponding word in its counterpart input, then the two inputs are considered to be identical text strings, and one is merely the duplicate of the other. If either condition is not met, i.e.:

1) The two inputs have different numbers of words

OR

2) Any word in an input is different (even in spelling) from the corresponding word in its counterpart input,

then the two inputs are considered to be distinct.

## Results

As a basic trial of the program, consider the following two inputs:

1) THE SNOWMAN WALKS ALONE STOP

vs.

2) THE SNOWMAN WALKS ALONE STOP

Input 1) has exactly one space between each consecutive pair of words. Input 2) has three spaces between each consecutive pair of words. Figure 1 displays the output from this comparison using the code in the Appendix. Note that a word-by-word comparison of the two inputs was conducted, and each word was evaluated as identical to its counterpart in the other input, regardless of the spacing differences. Therefore, even though there is a significant difference in the spacing between the two inputs, the comparison code deems the two input strings overall (i.e., in the final line of the output) to be identical. This evidences the success of the program in ignoring superfluous spaces between words.

In order to demonstrate the word count comparison feature of the code, the following two inputs were provided:

1) THE SNOWMAN WALKS ALONE STOP

vs.

2) THE SNOWMAN WALKS ALONE NOW STOP

The only difference between the two strings is the addition of the word “NOW” at the end of the sentence. Figure 2 displays the output from this comparison. A word count is conducted, and the program correctly concludes that the two text strings contain different numbers of words. On this basis, the overall conclusion (i.e., the final line of the output) is that the two text strings are different. Note that a word-by-word comparison is not conducted, since the word count is a more “global” (i.e., less detailed) measure by which to compare the two strings. The mechanism of the program is to compare word counts first, and if they are equal, to conduct a word-by-word comparison. If the word counts are not equal, the word-by-word comparison is omitted and a conclusion of inequality is made solely on the basis of the word count. This will be examined in more detail in the Discussion Section (*vide infra*).

Another consideration is that the word count is less computationally intensive than a word-by-word comparison, so omitting the word-by-word comparison for text strings of unequal length introduces a measure of computational efficiency. Thus, this test evidences the success of the program in identifying differences in lengths of input strings.

In order to evaluate the ability of the program to detect differences in individual words, the following two inputs were provided:

1) THE SNOWMAN WALKS ALONE STOP

vs.

2) HTE SNOWMAN WALKS ALONE STOP

This comparison highlights a misspelling commonly found in cable traffic. A typist intends to type the word “THE” and instead types the pseudo-word “HTE”, thereby introducing a typographical error to the hard

copy. This error subsequently is scanned into a text file, and comparisons with a text input string without this typographical error, but otherwise identical, should be detected as a single-word difference. An error of this nature represents one of the smallest differences that can exist between two text strings, since the word counts are the same, the number of characters in each word also displays no variation, and the spacing is identical.

The output from this comparison is displayed in Figure 3. The word counts are the same, so a word-by-word comparison is conducted. The program correctly concludes that there is a difference between the two inputs' first words (each referred to as Word # 0, due to the indexing methodology in the word stack; "SNOWMAN" is the second word, and is referred to as Word # 1, etc.). Each subsequent word is correctly evaluated as identical to its counterpart. Finally, the overall conclusion (i.e., the last line in the output) is that, based on the difference in one or more words, the two strings are different.

## Discussion

The functionality of each part of the code in Appendix A illustrates some fundamental concepts that apply to an object-oriented methodology. The major difference between a procedural methodology and an object-oriented methodology is that an object-oriented methodology allows a program to designate objects that are members of a pre-defined class, and then uses class member functions to manipulate the objects (Stroustrup, 1991). In contrast, a procedural methodology calls procedures to manipulate numbers in a variety of functions, but does not automatically associate a set of variables with each other (or with an object, as in the case of object-oriented programming).

An object-oriented approach to programming focuses on data and behavior that relates to the data, as opposed to procedural methodologies, which focus on means to manipulate data. In object-oriented programming, data and functions processing the data are considered classes whose instances are objects. Objects are variables belonging to a class (generally a user-defined class). Related to this is the concept of an Abstract Data Type (ADT), which can be considered as a user-defined extension to the base data structures provided by a high level language package. An ADT comprises a

set of values and a group of functions for which the data represent the domain of said functions.

OOP embodies a number of properties that are designed to promote facile implementation of ADTs. One of the most useful of these is the concept of inheritance, in which a new data structure can be derived from an existing one. As a case in point, text structure is in many respects similar to this mechanism. For example, words and phrases are types of text strings. If the information encoding text strings applies to all instances of that class, then creating the class “word” from the class “string” (i.e., creation of a subclass) represents an avoidance of duplication of effort.

In OOP, classes need to define not only data structures (objects, in particular), but also the operations that can be carried out on such objects. For example, if a text string is an ADT, then user-defined operations must be encoded for functions such as concatenation, redaction, and testing for equality. These are referred to as class member functions, in that they are specific to objects, or instances of a given class. Programming should be provided during class definition for any operations that objects could be expected to experience. This approach is demonstrated by the code in the Appendix. The class member functions include a concatenation function (to join two data members) and a destructor function (to deallocate memory associated with an object), even though these class member functions were not utilized in the main body of the program. At the time that the class was formulated, it was deemed possible that these functions would be needed, so the class includes them.

Another feature of OOP is the concept of encapsulation. Some operations are unavailable to objects or other data structures outside a particular class. Encapsulation includes the internal implementation details of a particular type, as well as the externally available operations and functions that can act on objects of that type. Details regarding the execution of certain operations can be hidden from user-provided code that implements the data type. For example, if a program to calculate masses or lengths uses a class member function to convert units from English to metric units, changing the internal workings of the unit conversion code should have no effect on the accuracy of the calculation. In short, the implementation of the unit conversion is hidden from its clients.

OOP can require a more sophisticated understanding of programming procedures than top-down programming or procedural methodologies. In other programming methodologies objects generally are implicitly created, and subsequently destroyed after being used. In contrast, C++ offers user

control of dynamic memory allocation. Selection and definition of objects in C++ allows memory to be apportioned for the object and its associated data. (If a data structure or object is not used at some point in the program, memory can at that juncture be deallocated as well.) Thus, selection of objects is a key to a number of facets in an OOP approach to problem-solving.

The code in the Appendix merits closer examination to highlight the functionality of certain portions of the class and the main program. Lines 3 through 8 consist of class definitions and inclusion of files for input/output, standard library functions, mathematical functions, and string functions.

The String class (lines 10 through 75) consists of data associated with each object that is an instance of the String class, and a number of class member functions. The data includes private data members for a pointer to a character, and the length of the object; and a public String function defining a null string.

The first String constructor function (lines 20 through 24) takes as its argument a pointer to character data, assigns memory for a new string of appropriate length, and copies the characters to the new string. The second String constructor function (lines 26 through 30) is similar to the first, but simply makes a copy of an existing object.

The set function (lines 32 through 39) changes the contents of existing character data by substituting into the data memory the string pointed to by the function argument.

An overloaded equality operator (lines 41 through 43) takes a string by reference and copies another string to the string passed as an argument.

The \*get function (lines 45 through 47) simply returns a pointer to data.

Lines 49 through 51 is an equality function that compares two strings, returns a “true” (or nonzero) value if the strings are the same, or returns “false” (or zero) if the strings are different.

A logical equality operator (defined in Lines 53 through 55), denoted as == (two consecutive equals signs), can be used in Boolean expressions, and builds on the equality function in lines 49 through 51.

A concatenate operator is defined in lines 57 through 65. It is denoted as +=, and it takes an existing string of character data, and appends additional character string data onto the end of the original string. It uses the library functions strcat and strcpy in the implementation of the class member operator.

A print function (lines 67 through 70) displays the contents of the data, and (with the optional line shown as line 69) can be used to give the string length of the data being displayed. As mentioned previously (vide supra), the stack approach to character data does not include spacing between words as characters in the stack. This is verified when running the program with line 69 included, as opposed to the version listed in the Appendix, where it has been Rem-ed out (or, in C++ vernacular, “// -ed out”). With line 69 included, the string length of any word is calculated using the library strlen function, and printed out alongside each word. Thus, it is easy to verify there are no spaces among the stack data.

The final class member function is the destructor function. When there is no further need for data to be stored, the memory can be deallocated, thereby adding computational efficiency.

The main body of the program begins with line 78. Two String objects, referred to a stack and stack2, are declared as variables, and a Boolean variable called dup is declared and initialized to unity (the default value for equality between stack and stack2). Any comparison that indicates a difference between stack and stack2 will set dup to zero, and result in an overall conclusion that the two strings are different.

Integers called topofstack and topofstack2 are initialized to zero, which signifies that, initially there are no data members in either stack.

Lines 89 through 97 solicit input data, and add the data to stack. A single line of text, consisting of up to 81 characters, can be input to the stack in a single line of input. A message directs the user to input data, and as long as the user does not input the tilde character (~), more words are added to the stack. Input is considered to be individual consecutive characters, with the exception of the space character and the tilde. The space character (actually, one or more consecutive space characters) tells the program to end the input of a word to the stack, and begin the input of the next word; the tilde character tells the program to stop adding to the stack. Line 96 calls the set function to add data to the stack, and increments the topofstack variable so that the next word added to the stack will go in the following memory location. Lines 99 through 107 conduct a similar input process on stack2. This program calls for character data to be input manually, but applying the code in the Appendix to the task at hand will result in the input being from a scanner via OCR software. This point, although minor, merits being mentioned.

Line 109 compares the two stacks for equality of word length. If there are an equal number of words in the two stacks (i.e., if topofstack ==

topofstack2) [N.B., again, a double equals sign is used to denote a Boolean comparison], then each word in stack is compared to its counterpart in stack2 (see line 113). If there are different numbers of words in the two stacks (i.e., topofstack is different than topofstack2) then the word-by-word comparison is skipped, and the Boolean variable is set to zero (line 133). Line 132 denotes that a difference in word count has been detected.

The word-by-word comparison is conducted in line 113. (The 'for' statement in line 111 moves the comparison from one word in the stack to the next.) The Boolean equality operator, as defined in lines 53 through 55, is employed in this section. If any word in stack is identical to its counterpart in stack2, then that result is printed (line 115) along with the word (line 116). If any word in stack is different than its counterpart in stack2, then the two words are printed (lines 122 through 124), and the Boolean variable dup is set to zero (line 126).

Both stacks then are printed out in full (lines 138 through 144). Finally, the variable dup is evaluated. If no condition was encountered that would indicate a difference between the two inputs, then the value of dup is expected to have been unchanged during the running of the program, and would have retained the value of unity. That overall result is printed (line 147). Otherwise, the overall result is that a difference was encountered, and the variable dup has been set to zero. The output corresponding to that situation is in line 149.

Several central themes in object-oriented modeling and design have been demonstrated with this work.

\* Abstraction consists of focusing on essential behaviors of an object or class of objects, while ignoring incidental properties. Abstraction, when applied to the design of classes and objects, denotes the process of identifying the potential uses and manipulations pertinent to a hypothetical object before considering how implementation should occur. This approach was put to use in designing the String class, since a number of potential class member functions were designed prior to the class and any associated objects being implemented.

\* Encapsulation is the process of separating the external aspects of an object, which are accessible to operations outside the class, from the internal workings of a class, which are hidden from all entities external to the class. Encapsulation prevents code from becoming so interdependent that small changes can have a ripple effect and impact far-flung portions of the program that were beyond the designer's intentions. The String class developed

herein uses encapsulation to good effect by providing class member functions that control access to object data, while protecting the data most vital to each object (specifically, the memory location and the string length).

\* Polymorphism is the ability to use similar operations in different contexts. For example, the String class defines an equality operator to be used to compare two strings, and a concatenate operator ( $+=$ ) that appends one string to another in order to form one longer string.

\* Inheritance is a property that was not needed for this single- class system, but may be employed in future work that will build on that presented herein.

\* Reusability is related to encapsulation in that avoiding interdependence of code allows portions of code to be used in new applications with minimal editing. The importance of this property is evidenced by the code in the Appendix having been adapted from previous applications, and from adaptation of code from other object-oriented designs to future applications (Phillips, et al., 1999).

## Conclusions

We find that work has broken new ground in applying principles of object-oriented programming to semiotic modeling. The advantages of an object-oriented approach are intertwined with those of using a word stack approach. There are a number of benefits associated with the code in the Appendix relative to code developed for string processing using previous approaches, most of which were procedural, as opposed to object-oriented. Future work is expected to expand upon these benefits and advantages, and apply them to even more challenging problems.

## References

- Aitken, P. G. *Visual Basic for Windows 95 Insider*, (Wiley; New York; 1996) 1ff.
- Below, R. K., "Adaptive Information Retrieval: Using a Connectionst Representation to Retrieve and Learn About Documents," in Belkin, N. J., (Ed.), *Proc. 12th Annual International ACM SIGIR* (1989).
- Bouchaffra, D., Meunier, J.-G., "A Markovian Random Field Approach to Information Retrieval," *Conference Internationale de Documente L'Analyse et Retrievale (Compte Rendu)*, 11, 997 (1995).
- Capper, D. M., *C++ for Scientists, Engineers, and Mathematicians* (Springer-Verlag; London; 1994) 6.
- Gilbert, J. D., *Elements of Linear Algebra* (International Textbook Company; Scranton, PA; 1970) 74.
- Kuratowski, K., Mostowski, A., *Set Theory* (PWN – Polish Scientific Publishers; Warsaw; 1968) 171.
- McMonnies, A., McSporrán, W. S., *Developing Object-Oriented Data Structures Using C++* (McGraw-Hill; London; 1995)132ff.
- Meunier, J.-G., "The Categorical Structure of Iconic Languages," *Laboratoire LANCI Home Page* <http://pluton.lanci.uqam.ca/membres/meunier/iconlang/iconlang.htm>. (1998).
- Meunier, J.-G., Bertrand-Gastaldy, S., Lebel, H., "A Call for Enhanced on Text Content for Information Retrieval," *International Classification*, 5 (1987).
- Meunier, J.-G., Bertrand-Gastaldy, S. Paquin, L. C., "La gestion et l'analyse de textes par ordinateur : leur specificité dans le traitement de l'information," *Revue de Liaison de la Recherche en Informatique Cognitive des Organisations (ICO Québec)*, 6(1 et 2) 19(1993).
- Michel, A. N., Herget, C. J., *Mathematical Foundations in Engineering and Science* (Prentice Hall; Englewood Cliffs, NJ; 1981) 12.
- Muller, G. H., *Sets and Classes* (North-Holland; Amsterdam; 1976)173ff.
- Partridge, D., Wilks, Y., eds., *The Foundations of Artificial Intelligence* (Cambridge University Press; Cambridge;1990) 47ff and 165ff.
- Phillips, J. B., Price, G., Fry, S., Arcziszewski, T., DeMonsabert, S., Menawat, A., "An Object-Oriented Approach to Numerical Methods:The Regula Falsi Method for Solving Equations with Tight Tolerances for Environmental Applications," *J. Haz. Materials* (In press, 1999).
- Prata, S., *C++ Primer Plus* (Waite Group Press; Corte Madeira, CA; 1995) 5.
- Rijsbergen, V., *Information Retrieval*, 2nd Ed. (Butterworth & Cie Publishers, Ltd; London; 1979) 1ff.

- Robertson, S., Harding, P., "Probabilistic Automatic Indexing by Learning from Human Indexers," *J. Documentation*, 40(4) 264 (1984).
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, (Prentice Hall; Englewood Cliffs, NJ;1991) 7-10.
- Sedgewick, R., *Algorithms in C++*, (Addison Wesley; New York; 1992) 1ff.
- Selby, S., Sweet, L., *Sets Relations Functions* (McGraw-Hill; New York; 1969) 88.
- Sodhi, J., Sodhi, P., *Object-Oriented Methods for Software Development* (McGraw-Hill; New York; 1996) 149ff.
- Stroustrup, B., *The C++ Programming Language*, 2d ed. (Addison Wesley; Reading, MA; 1991) 1ff.
- Walsh, A. E. *Java*<sup>TM</sup>, (IDG Books Worldwide; Foster City, CA; 1996) 1ff.
- William, M., *Connectionist Models and Information Retrieval*, 25, 209 (1990).