Translating English to Mathematics:
A Linguistic View of Mathematics for Software Engineers

Draft – readers' comments desired, please email to: Baber@McMaster.CA

an electronic monograph
by Robert L. Baber

2003 February 17

McMaster University
Department of Computing and Software

Table of Contents

1. Introduction

1.1 Mathematics in software engineering

Characteristic of every engineering field is a theoretical, scientific foundation upon which most of the engineer's work is based. An essential component is one or more models of the artifacts or processes being analyzed or designed. Mathematics is the *language* in which engineers continually, systematically, consciously and even subconsciously formulate, describe, discuss and reason about these models when designing and analyzing artifacts. This basis for engineering work is a prerequisite for achieving the reliability of designs to which engineers, their clients and society have become accustomed.

Thus, mathematics is the language of engineering. Software engineering is no exeception; mathematics is just as important in the daily life of a true software engineer as it is in the daily life of other engineers, perhaps even more important. A software developer may be able to get along without "speaking" mathematics fluently and actively, but a software *engineer* cannot.

Many software engineering students begin their studies with only a passive knowledge of mathematics, much like someone who can slowly read text in a foreign language, but who cannot speak or write an essay in that language. Because the software engineer needs an *active* ability in mathematics, the software engineering student must develop an active fluency in mathematics, and the sooner, the better.

1.2 The goal and role of this monograph

The goal of this electronic monograph is to help software engineering students to acquire, develop and improve their active ability in using the language of mathematics. As with comparable skills in the natural languages, developing one's active ability in mathematics requires practice and drill. No person, no book can teach you these skills; they can only help and guide you in *your* learning process. *You* must do the work. You must write your thoughts in mathematics. You must read, speak and listen to others speaking mathematics. You must translate problems stated in English into mathematics. You must express software specifications in mathematics. This monograph will help you to get started.

1.3 Non-goals of this monograph

This monograph is not about mathematics as a subject and it is not intended to help you learn mathematics itself. It does not deal with the various topics typically covered in books on mathematics. Passive knowledge of these subjects is a prerequisite for working through this monograph. When you encounter in this monograph mathematical topics with which you are not familiar, refer to an appropriate book on the relevant area of mathematics. A few such books are listed in the bibliography in Appendix F below.

1.4 Structure and contents of this monograph

The role and importance of mathematics in software engineering and the goal and scope of this monograph constitute the subject of chapter 1. The nature of languages – including mathematics – and translating between them, particularly into mathematics, is the topic of chapter 2. Chapter 3 gives examples of translating English texts into mathematical expressions and analyzes the steps involved in some detail. Generally useful principles, guidelines and techniques are identified in these examples. Chapter 4 examines several approaches for checking mathematical expressions for validity and plausibility. Chapter 5 summarizes and draws conclusions from the preceding material.

Appendix A is a glossary of selected English terms and the corresponding mathematical objects and symbols, collected from the examples worked out in chapter 3. Similarly, Appendix B collects strategies and tactics developed in the examples in chapter 3. Appendix C and Appendix D deal with selected aspects of mathematical notation as used in this monograph. Appendix E discusses alternatives for handling undefined terms in expressions. Appendix F is a bibliography of books and articles containing material on various relevant mathematical topics and containing exercises for the reader.

2. Mathematics as a language

2.1 Similarities and differences between languages

Mathematics is viewed in this monograph as a language. The natural languages such as English, French, German, Zulu, Shona, Sipacapa, Esperanto (not really a natural language in origin, but a natural language in type and style), Tzotzil, Xhosa, Akkadian, Plattdeutsch, Letzeburgisch, Latin, Arabic, Hebrew, Sanskrit, etc. have both common and distinguishing characteristics.

Almost all – perhaps all – natural languages have the same basic sentence structure: subject, predicate and sometimes an object. The basic types of words are also a common feature: nouns, pronouns, verbs, prepositions, adjectives and adverbs. Shortly thereafter, however, differences arise, e.g. the extent to which the form of one word in a sentence must agree with other words in the sentence, the ways in which such agreement is expressed, and the possibility of expressing meaning by the relationships established by such agreement. Furthermore, the vocabularies of different languages differ according to the environments and needs of the societies. For example, Arctic societies' languages typically have words and short expressions enabling one to talk about many more types and variations of snow, for example, than do languages spoken in areas with only little or no snow. Similarly, languages (e.g. Arabic) spoken by people living in sandy desert areas permit one to concisely make much finer distinctions between different types of sand, sand surfaces and sand formations than do languages spoken by societies in regions with little or no sand. Languages of societies that have only relatively recently come into contact with the western world typically lack words common in technologically advanced societies, e.g. words associated with such things as "electricity", "telephone", "computer", etc. Zulu, for example, has no own word for "radio" and the Zulu word for "shirt" and other common terms in western societies are borrowed words; even the sound and the letter "r" has entered the Zulu language only through borrowed words such as "radio".

One thinks somewhat differently in each language. One typically says different things in the different languages. Some thoughts and expressions are normal in one language and quite strange and unusual in another. These differences complicate translating between languages. Even when thoughts are the same they are often expressed differently in different languages and such differences must be taken into account when translating.

For example, "Yes, we have no bananas" is self-contradictory in English but is a literal translation of the normal way of answering the corresponding question in some languages. In those languages, the statement "No, we have no bananas" is self-contradictory.

Another often cited example begins "Throw Mama from the train ...". As soon as one hears this first part of the sentence in English, one already thinks of someone trying to kill the mother by throwing her out of a train. When a German speaker hears this first part of the sentence in German, it is quite clear that one must be throwing something *to* Mama and one expects the thing being thrown – e.g. "a kiss goodbye" – to be stated next.

These are only very simple examples of how people think differently in different languages. One must be aware of such differences in the way people think in different languages when translating text from one language to another.

Mathematics as a language differs from the natural languages in even greater ways. The differences begin already at the basic sentence level. Mathematics does not share the sentence structure (subject, predicate, object) of the natural languages, at least not explicitly. A basic unit in mathematics is an expression, which states a relationship – and a certain *kind* of relationship – between the several objects in the expression. If one thinks of an expression as equivalent to a sentence in English, the verb is implied and is usually "to be" (or a conjugated form such as "is" or "are") or a verb of state or being rather than a verb of action. Therefore, mathematics is a language of static statements; its world view is a static one. One can make statements in mathematics about physically dynamic processes, but the statements themselves are inherently static.

An especially important difference between natural languages on the one hand and mathematics on the other hand is their view and treatment of vagueness. All natural languages permit, quite intentionally, vague statements. It is essentially impossible in any natural language to make an absolutely unambiguous statement. Mathematics is, by intent and design, just the opposite: it is essentially impossible to make a vague or ambiguous statement in correct mathematics. This is both an advantage and a disadvantage of the language of mathematics, depending upon the purpose of discourse. For engineering work, the advantage of unambiguity is important. Especially when specifying software it is critical to avoid ambiguity and for this purpose the software engineer must express herself or himself in mathematics and avoid English. This difference cannot be overemphasized: In mathematics, one thinks and speaks unambiguously. In English, one thinks and speaks ambiguously. No matter how hard one tries to express oneself unambiguously, a remnant of ambiguity, however small, is always present in natural language text.

The differences among natural languages often have environmental and cultural origins (as already alluded to above) and such differences have, over time, become deeply ingrained in the respective languages. Again, in the case of mathematics, this observation applies. The culture of mathematics (unambiguity and logical reasoning) is reflected in the language of mathematics and in the ways that language is used. Culture, the way of thinking, mentality and the view of the world, and language are inseparably related, no matter what the language is. Mathematics is no exception to this observation.

In summary, a language is not just a means of communication. It is also a way of thinking, a view of the world, a culture. To translate successfully, one must be able to translate between ways of thinking, between views of the world, between cultures.

2.2 The process of "translating" or "interpreting" between languages

In order to "translate" something expressed in one language into another language, one must

1. read or listen to the message in the original language,
2. understand (internalize) the meaning of that message in the original language and in the context of the subject of the message,
3. understand (internalize) that meaning in terms of the target language and then
4. write or speak something in the target language

so that a reader or listener of the message in the target language will interpret it to mean the same thing that a reader or listener of the message in the original language would interpret it to mean.

After one believes that the translation is complete, one must then

5. reread and understand (interpret again) the message in the original language,
6. read and understand (interpret anew) the translated version in the target language and
7. compare the meanings of the two.

If any discrepancies are discovered, they must be resolved by repeating some or all of the above activities as appropriate. An important discrepancy arises if, in either step 1 or 2, it becomes evident that the original text is inconsistent, incomplete, clearly wrong or makes no sense. One should always be sensitive to these possibilities, as they arise in software engineering practice more often than one might expect.

Note especially that "translating" is *not* a matter of going immediately from activity 1 above to activity 4. The activities 2 and 3 above – understanding in the two different contexts – is essential. Often, iteration is required to complete activities 2, 3 and 4 above. An independent check is also needed (activities 5, 6 and 7 above). If at all possible, one should wait a while between activities 4 and 5, because as long as the material is fresh in the translator's mind, the brain will often fill in missing detail and subconsciously correct errors present. Also desirable is that a different person perform activities 5, 6 and 7 than the person who performed activities 1, 2, 3 and 4.

The English word "translate" is often used to describe this process, but "translate" suggests an oversimplified view of what actually must take place. The word "interpret" is more suggestive of activities 2 and 3 above – understanding the meaning of the message before writing or speaking it in the target language. In fact, professionals who "translate" from one language to another, especially simultaneously and verbally, often call themselves "interpreters", not "translators", and their work is often called "interpreting", not "translating".

Attempts to translate material from a source language to a target language without going through the intermediate activity of understanding are typically unsuccessful. The true meaning of the original is not properly and correctly conveyed in the target language. The most recent example of this I encountered was a piece of German text translated by machine into English. A person's title and name "Prof. Dr.-Ing. Hans-Jürgen Hoffmann" was translated as "Professor Dr. Dr.-Ing. Hans Juergen hoping man". The misinterpretation of the surname "Hoffmann" as a compound noun led to "Hoffmann" being translated as "hoping man", a reasonable literal translation given the misunderstanding. Perhaps the compound given name "Hans-Jürgen" – a structure common in German culture – was misinterpreted as a given name followed by a surname (Jürgen), so that the following word was not recognized as a surname. In this example one can see how the lack of knowledge of the culture of the original language can lead to significant errors in the meaning of the purported translation in the target language. As a consequence, one can appreciate the importance of interpreting and understanding the original text in the context of its culture in order to obtain an accurate rendition in the target language.

Another consequence of the observation above that understanding is necessary is that fluency in both the original and target languages is not enough for successful translation. The translator must be able to *understand* the material in the original language, to recognize the meaning of the material. Therefore, if one is to translate a scientific article in a journal on chemistry from French into English, one must be fluent in both French and in English and, in addition, one must have some knowledge of chemistry. Furthermore, knowledge of translating as an activity itself is also necessary for efficient work. Therefore, professional translators and interpreters are often graduates of several year specialized educational programmes in translating.

Certain terms or phrases in one language often correspond to particular terms or phrases in another language. People frequently translating between languages often collect a set of such corresponding pairs of terms or phrases and record them in their *translator's glossary*. This also applies when one of the languages is mathematics.

In summary, in order to interpret English into mathematics, you must

- be fluent in English
- be fluent in mathematics
- know the subject about which you are interpreting statements from English into mathematics,
- be familiar with certain terms and phrases typically corresponding with each other in English and mathematics and
- be consciously aware of the special aspects of the process of interpreting from one language into another, especially into mathematics.

This monograph is intended to help you increase your skills in the last two areas above.

2.3 A dichotomy in the world views of mathematics and software

Many persons who develop software tend to think primarily in terms of programs or algorithms – dynamic processes in which certain changes are made in each step in a sequence. Their world view concentrates on the passage of time and on change.

Mathematics, in contrast, is basically a language dealing with static views. Mathematical objects – e.g. relations, functions, formulae, expressions, etc. – do not change, they are constant. Even dynamic processes in the physical world are modelled with static functions, one parameter of which represents time. In other words, mathematics typically maps a dynamic view in the application domain into a static view in the mathematical domain.

This discrepancy often leads to confusion on the part of software students. For example, when trying to write a mathematical expression specifying the state of a system after a program has executed, students frequently tend to write an expression that reflects the temporal process of a program execution that computes the final state. They sometimes confuse a quantified expression in mathematics with a loop in a program that calculates values of variables appearing in the quantified expression. Often such confusion leads to a mathematical expression that contradicts itself and is simply wrong. One must distinguish rigorously between the static view given by the mathematical expression and the dynamic view represented by a program. One must interpret from the one view to the other in the mental process of translating, in particular, in the activities 2 and 3 described in section 2.2 above.

3. Examples

In each of the examples below, a task is described in English. Some aspect of the English description is to be translated into mathematics. The English description is then transformed in a series of steps into the desired statement(s) in mathematics.

In the process, guidelines and principles will be identified, typically in the form of a *strategy* (a general goal of the guideline or principle) and a *tactic* (a specific way of achieving the goal). The strategies and the tactics are *not* universal principles and techniques, always applicable to *all* problems. Each is useful in many – but *not* all – situations. Select those which help as needed for a particular task, but do not attempt to apply any of them just for the sake of applying it. These strategies and tactics will be collected and reorganized in Appendix B below.

Some terms in English typically correspond to particular objects, functions, terms, etc. in mathematics. Such corresponding terms will be identified and collected for the translator's glossary in Appendix A below.

3.1 Searching an array

3.1.1 Simple search

Task: Consider an array D with index values ranging from 1 to n. The subject of this example is part of a specification for a subprogram that will find the location in D of a particular given value. The goal of this exercise is to write a mathematical expression describing the final state of the search, i.e. a relation between the various relevant variables' values when the subprogram finishes and the search is complete.

The first activity in the process of translating from the above description in English to the expression in mathematics (see section 2.2 above) is to read the above text in English. Then, that description must be understood in English and in terms of the array and the search.

Note below how the various ambiguities in the English text are identified and resolved in order to obtain the information needed to formulate the unambiguous mathematical expressions.

Strategy: Understand the meaning of the message.

Tactic: Identify missing information.

Tactic: Identify implicit – but not necessarily true – information.

Tactic: Identify special cases, especially ones not explicitly mentioned.

Sometimes needed information is not given in the English statement of the problem. Often special cases are not explicitly mentioned in an English description of the task and they are, therefore, overlooked. These discrepancies can lead to an incomplete translation in mathematics and consequent problems that can be difficult and time consuming to resolve.

In order to find a "particular given value", that value must be known. Within the context of a computer program, it will presumably be given in the form of the value of a particular variable.

Stop. Question: Is this a reasonable presumption? Why or why not? What other alternatives could be used?

The name of the variable will be needed, but is not given in the English text above. Normally, the person writing the mathematical specification will ask the person who wrote the above task description for the missing name of the variable. We will assume that the answer is “key”.

Strategy: Obtain all needed information.

Tactic: Ask the author of the task description.

Stop. Question: Scan the English text again. What variables are mentioned? Does the text give us all possible information about them? If not, what is missing? Do we really need all of the missing information?

The English text above explicitly mentions the array D and the variable n. It says nothing about the type of data values in D, i.e. whether they are integers, “real” numbers, strings, elements of some other specified set, single values or compound structures, etc. In each of these possible cases, additional questions about the properties of the data arise, e.g. how small and how large the numbers can be, how long the strings can be, etc. In the case of the variable n, similar questions arise, in particular, whether n may be negative or zero, how large n may be, whether or not n must be an integer, etc.

Strategy: Obtain all needed information.

Tactic: Identify gaps in the description of the task.

Tactic: Read carefully, thoroughly and precisely.

Tactic: Identify the data present and ask questions about related details.

Tactic: Question whether missing information is really needed.

Tactic: Identify implicit “information”.

Tactic: Question explicitly whether implied information may be assumed.

Implicit in the English description above of the task is that the value being sought is actually in the array. Sometimes this can be guaranteed, but more often than not, the value being sought cannot be guaranteed to be in the array. In the latter situation, which we will assume here, we distinguish between the two cases in which (1) the value being sought is present in the array D and (2) the value being sought is not present in the array D.

Stragegy: Divide and conquer.

Tactic: Distinguish between specific cases.

I.e. we distinguish between case (1):

$(\vee\, i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key)$ [case (1): value of key present in array D]

and case (2):

$\neg(\vee\, i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key)$ [case (2): value of key not present in array D]

The latter is equivalent to

$(\wedge\, i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) \ne key)$ [case (2): value of key not present in array D]

We must now decide what is desired in each case.

When the value being sought is in array D, the above task description states that the subprogram must find the location in D of a particular given value. I.e., that location must be known and presumably made available to subsequent parts of the program. Again, a variable will presumably be needed for this purpose, so we must ask the author of the task description the name of the variable. We will assume that the answer is "location". I.e., in this case, D(location) = key (and the value of the variable named "location" will be an integer between 1 and n inclusive).

If the value being sought is not in array D, the above task description says nothing about what the state of the various variables should be on completion of the subprogram. Again, the author of the text should be asked what he meant in this case. We will assume that the answer is that the value of the variable location – the name of the output variable in the first case above – should be outside the possible range in the first case above. E.g., the value of location might be n+1. For the sake of simplicity, we will require that the value of location be n+1.

Stragegy: Divide and conquer.

Tactic: Construct a table.

We combine the above answers and assumptions and present them in the form of a table:

Case (in English):	case (1): key present in array D	case (2): key not present in array D
Case (in mathematics):	$(\vee\, i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key)$	$\neg(\vee\, i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key)$
expression:	$location \in \mathbf{Z} \wedge 1 \le location \le n$ $\wedge\ D(location) = key$	$location = n+1$

The reader should now reread the English statement of the problem above and compare it with this table. Check word by word, phrase by phrase, etc. for consistencies, inconsistencies, errors, omissions, etc.

The desired expression is given unambiguously by the above table and can either be left in this form or written in an equivalent but more traditional form, e.g.:

$$[(\vee\, i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key) \Rightarrow (location \in \mathbf{Z} \wedge 1 \le location \le n \wedge D(location) = key)]$$
$$\wedge\ [\neg(\vee\, i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key) \Rightarrow (location = n+1)] \quad [1]$$

which can be rewritten in any of several equivalent forms, e.g.

$$[\text{location}\in \mathbf{Z} \wedge 1\le\text{location}\le n \wedge D(\text{location}) = \text{key}]$$

$$\vee\ [\neg(\vee\ i : i\in \mathbf{Z} \wedge 1\le i\le n : D(i) = \text{key}) \wedge (\text{location} = n+1)] \qquad [2]$$

Stop. Mathematical exercise: Prove that the expressions [1] and [2] above are equivalent. Why is there no quantified subexpression in line 1 of expression [2]? Hint: This is an exercise in manipulating Boolean algebraic expressions; no other analysis is needed.

3.1.2 Finding the first value in an array

Task: This task consists of the same requirements as the task in section 3.1.1 above with the added requirement that the element of D found to be equal to key must be the first such element in D.

Stragegy: Divide and conquer.

Tactic: Modularize.

Tactic: Use results of previous tasks as components of solution to this task.

Translating the above text into mathematics can be done by splitting the task into two parts, one of which has already been solved, and adding the additional requirement. Therefore, we concentrate first on the additional requirement.

The additional requirement is that the element of the array found to be equal to key, i.e. the element D(location), be the first such element in D. The question then arises what is meant by “first”. Probably this is to be interpreted that no element of D with a lower index value than location is equal to key. Implicit in this reasoning is the assumption that D(location) = key, i.e. that there is at least one element of D with the value key. I.e., this analysis applies only to case 1 in the solution to the search in section 3.1.1 above.

The question then arises what the additional requirement means if case 2 applies, i.e. no element of D is equal to key. The most reasonable interpretation of the phrase

> “with the added requirement that the element of D found to be equal to key must be the first such element in D”

in the task above is that the condition “if any” is implied:

> with the added requirement that the element of D found to be equal to key – *if any* – must be the first such element in D

Adding the phrase “if any” to our intrepretation makes it clearer that in case 2 (no element of D is equal to key), the added requirement means nothing. I.e. in this case, the added requirement is redundant, unnecessary and irrelevant and can, therefore, be disregarded.

Note how the English text has been examined in detail and vagueness and ambiguity identified and reduced in order to bring it closer to the world view of mathematics, i.e. more precise and less ambiguous.

Strategy: Close the gap between the English text and mathematics.

Tactic: Reword the English text to bring it closer to mathematics.

Tactic: Reduce vagueness and ambiguity.

Tactic: Express implicit information explicitly.

We now return to the additional requirement and interpret it within the context of case 1 (at least one element of the array D has the same value as the variable key), see above. D(location) will be the *first* element of D with the value key if

- D(location) = key (this condition is already part of the solution to the simple search in section 3.1.1 above) and
- all elements of D with an index less than location are different from (unequal to) key.

Therefore, if we take the solution to the simple search in section 3.1.1 above and impose (in case 1 only) the second bulleted condition above, we will have a solution to our current task.

Thus, we have reduced the current task to the subtask of translating

“all elements of D with an index less than location are different from (unequal to) key”

into mathematics. The word “all” suggests an and-series (universal quantification).

Translator’s glossary: each, every, all, any ↔ for all, and (universal quantification, ∀, ∧).

Therefore, we translate the above phrase literally from English into

$(\wedge\ i : i \in \mathbf{Z} \wedge 1 \le i \le n \wedge i < location : D(i) \ne key)$

Combining our solution to the simple search in section 3.1.1 above with this added condition in case 1 only, we obtain

Case (in English):	case (1): key present in array D	case (2): key not present in array D
Case (in mathematics):	$(\vee\ i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key)$	$\neg(\vee\ i : i \in \mathbf{Z} \wedge 1 \le i \le n : D(i) = key)$
expression:	$location \in \mathbf{Z} \wedge 1 \le location \le n$ $\wedge\ D(location) = key$ $\wedge\ (\wedge\ i : i \in \mathbf{Z} \wedge 1 \le i \le n$ $\wedge\ i < location$ $: D(i) \ne key)$	$location = n+1$

which can be simplified to

Case (in English):	case (1): key present in array D	case (2): key not present in array D
Case (in mathematics):	$(\vee\ i : i\in \mathbf{Z} \wedge 1\le i\le n : D(i) = key)$	$\neg(\vee\ i : i\in \mathbf{Z} \wedge 1\le i\le n : D(i) = key)$
expression:	$location\in \mathbf{Z} \wedge 1\le location\le n$ $\wedge\ D(location) = key$ $\wedge\ (\wedge\ i\ : i\in \mathbf{Z} \wedge 1\le i<location$ $: D(i)\ne key)$	$location = n+1$

The expression represented by the above table can be expressed in more traditional form as in section 3.1.1 above. Among the equivalent expressions (assuming that n is an integer, which seems to follow implicitly from the original statement of the task) is

$$location\in \mathbf{Z} \wedge 1\le location\le n+1 \wedge (\wedge\ i : i\in \mathbf{Z} \wedge 1\le i<location : D(i)\ne key)$$
$$\wedge\ [(location\le n \wedge D(location) = key) \vee (location = n+1)]$$

3.2 Students with the same birthday

Task: Consider a class with many students. The instructor bets the students that two or more students have the same birthday. Write a mathematical expression that is true if this condition is met and false otherwise.

Again we must begin by reading and understanding the above text in English. Then we must convert the statement into a mathematical view. Finally, we must formulate the message in mathematics.

Strategy: Understand the meaning of the message.

Tactic: Identify all objects referred to in the message.

Tactic: Distinguish between essential objects and background information.

To identify all objects one should begin by looking at the nouns and pronouns in the English text. In this case, we have: class, students, instructor and birthday. Of these, the condition to be expressed in mathematics clearly involves students and birthday. It is not stated explicitly that this condition involves the class also, but this seems to be suggested. Presumably the phrase “two or more students have the same birthday” means “two or more students *in the class* have the same birthday”, in which case the condition involves the class. If this is not the intended meaning of the task description, the question arises which group of students is meant, the entire student body in the school in question, all students in the world, etc. The condition to be expressed in mathematics does not seem to involve the instructor, who is proposing the bet but who is not part of the condition being betted upon. We conclude, therefore, that the instructor is not one of the objects to be mentioned in the condition.

The essential objects for the condition to be written in mathematics are, therefore: class, students, and birthday.

The question also arises whether or not the word “many” is essential, i.e. whether it is part of the condition to be expressed in mathematics or whether it is general background information not of direct concern to the translator. We will assume the latter here. If “many” were significant, this would suggest that we must include a term in the resulting mathematical expression to the effect that the number of students must be greater than some constant (an implicit object), raising in turn the question what that constant should be. This latter uncertainty in the English text is another reason for assuming that the word “many” is not significant.

Strategy: Understand the meaning of the message.

Tactic: Identify relationships between essential objects referred to in the message.

To identify relationships between the essential objects we begin by considering all combinations of two or more of these objects. These combinations are:

- class, students
- students, birthday
- class, birthday
- class, students, and birthday

Regarding the relationship between the objects class and students, the text begins “Consider a class with many students”. In mathematical vocabulary, the reader would understand “consider a set (a class) of elements (the students)”. Renaming “class” as C and a student as S, the relationship becomes, in mathematics, $S \in C$.

Regarding the relationship between the objects students and birthday, we “know” from the cultural context of human society that each student is a person and that each person has a birthday. Furthermore, each person has a unique birthday, so that in the mathematical world view birthday is a function of the person in question, i.e. the student. Renaming “birthday” as bd, the relationship becomes, in mathematics, bd(S).

Regarding the relationship between the objects class and birthday, there appears, again from the cultural context of human society, to be no direct relationship. One could consider the set of birthdays of the students in the class, e.g. $(\cup\ S : S \in C : \{bd(S)\})$, but this is not explicitly referred to in the English text and from the statement of the task does not seem to be needed.

Regarding the relationship between the objects class, students, and birthday, there does not appear to be any direct relationship between all three of these objects; only the relationships above between pairs of these objects seem to be relevant.

Now we can turn to the condition to be expressed in mathematics. The English version of this condition is, after our above addition to it “two or more students in the class have the same birthday”. Thinking in terms nearer mathematical vocabulary, we can reexpress this condition as “for some student s1 in the class and some *other* student s2 in the class, bd(s1)=bd(s2)” (whereby we do not exclude the possibility that still other students have the same birthday). This sentence is in English syntax, English word order and mixed English and mathematical vocabulary, so we must finish by reexpressing it once more into purely mathematical syntax, purely mathematical subexpression order and purely mathematical vocabulary:

$$\boxed{(\vee\ s1, s2 : s1 \in C \wedge s2 \in C \wedge s1 \neq s2 : bd(s1)=bd(s2))}$$

Translator's glossary:	some $\leftrightarrow \exists, \vee$
	in $\leftrightarrow \in$
	other $\leftrightarrow \neq$
	same $\leftrightarrow =$

At this point the reader should review the mathematical expression above and verify that it is a correct translation of the condition in English into mathematics. Note especially that it properly handles the phrase "*or more*" in the English condition "two or more students in the class have the same birthday".

Potential pitfall: The devil lies in detail. Notice especially carefully in the paragraph above that the phrase "two ... students in the class" was reexpressed using the word "*other*" in the phrase "for some student s1 ... and some *other* student s2". In this way the implicit but clear fact that different students are meant in English by "two" has been made explicit in the mathematical formulation. Without careful attention to such detail we might have left out the essential term "s1≠s2" in the solution above. Without this term, the expression would be wrong and it would be a complicated way of writing the logical constant true. (Why would it always be true?).

Insufficient attention to detail is a common cause of errors in translating from English to mathematics.

3.3 Initialization of a game board

3.3.1 Initialization of a game board: a correct solution

Consider a game whose board consists of bowls, some of which contain stones. The players, who sit around a table, are numbered from 1 to NP counterclockwise. In front of each player are NB ordinary bowls placed in a line from left to right and one special bowl to their right. Initially each ordinary bowl must contain 5 stones and each special bowl must be empty.

A computer program for playing this game is to be designed. The subject of this example is part of the specification of the subprogram to initialize the board. The number of stones in each bowl will be stored in a one dimensional array named "StonesInBowl" with index values ranging from 0 to NP*(NB+1)-1. (Why is this formula for the highest index value correct?) The goal of this exercise is to write a mathematical expression describing the initial state of the bowls. The mathematical expression to be written will be an important part of the specification of the subprogram to initialize the board.

The first activity in the process of translating from the above description in English to the expression in mathematics (see section 2.2 above) is to read the above text in English. Then, that description must be understood in English and in terms of the game board.

Strategy: Understand the meaning of the message.

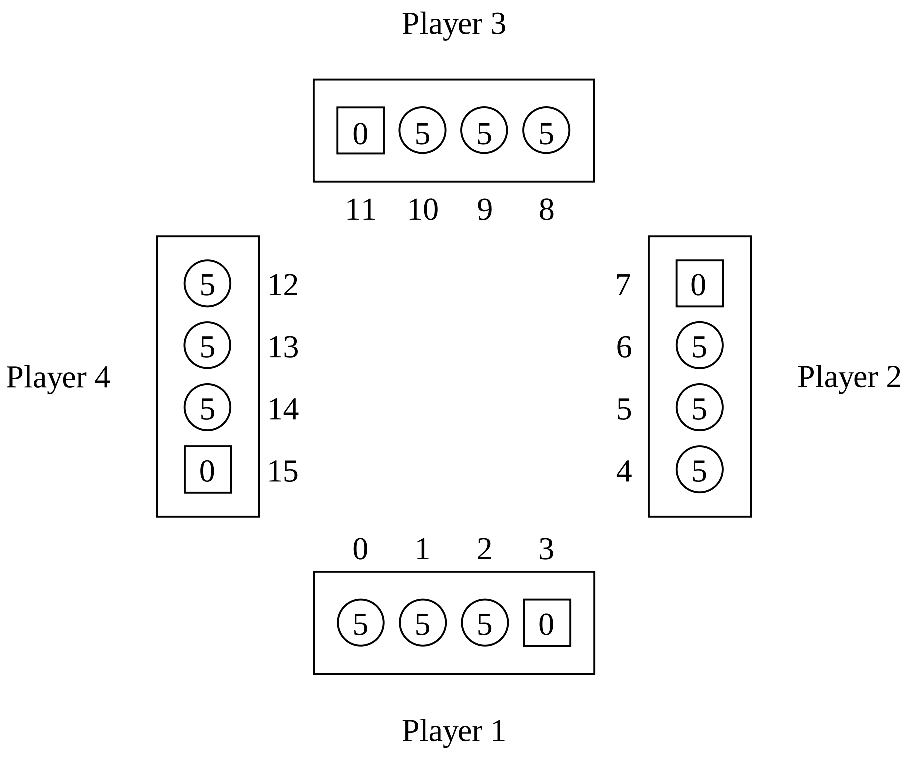
Tactic: Draw a diagram.

Because it is more difficult to draw a diagram for the general case of an unspecified number of players and of bowls than for a particular case, we will assume a specific number of players and a specific number of bowls.

Tactic: Describe a specific instance of the general problem.

To reduce the possibility of confusion later, a different value should be selected for each of the relevant parameters. Avoid also especially simple values such as 1.

We select NB=3 and NP=4 and draw the corresponding diagram below. The circles represent the ordinary bowls; the squares, the special bowls; the number in each bowl, the number of stones in the bowl; and the numbers in the open space in the middle, the indexes of the corresponding elements of the array StonesInBowl:



The reader should now reread the English statement of the problem above and compare it with this diagram. Check word by word, phrase by phrase, etc. for consistencies, inconsistencies, errors, etc.

Next, we must understand the meaning in mathematical terms. The text of the problem statement and the meanings of the various aspects of the diagram (e.g. the circles, squares, etc.) lead to mathematical expressions such as:

StonesInBowl(0) = 5
StonesInBowl(1) = 5
StonesInBowl(2) = 5
StonesInBowl(3) = 0
...
StonesInBowl(10) = 5
StonesInBowl(11) = 0
etc.

Each of the above expressions as well as *all* the other comparable terms must be true, i.e. they must be connected with the logical *and* function ($\wedge$). In the general case (for arbitrary values of NP and NB), we will need a *universal quantification* (*for all* construct).

Translator's glossary: each, every, all, any $\leftrightarrow$ for all, and (universal quantification, $\forall$, $\wedge$).

Because some of the bowls contain 5 stones and others, 0 stones, we will need two quantified subexpressions at the level of the elements of the array "StonesInBowl", one for the ordinary bowls and one for the special bowls.

The initial state of each player's collection of bowls is the same for every player. I.e., by considering first the several players (not the bowls), we need only one quantified expression, which will presumably be simpler than the double quantification needed if we would start at the more detailed level of the elements of the array StonesInBowl as outlined above. Then, within each player's collection of bowls, we would need another – but independent – quantification, leading to a simpler structure. We therefore apply another strategy to subdivide the one structurally complicated problem into two subproblems, each structurally simpler.

Stragegy: Divide and conquer.

Tactic: Modularize.

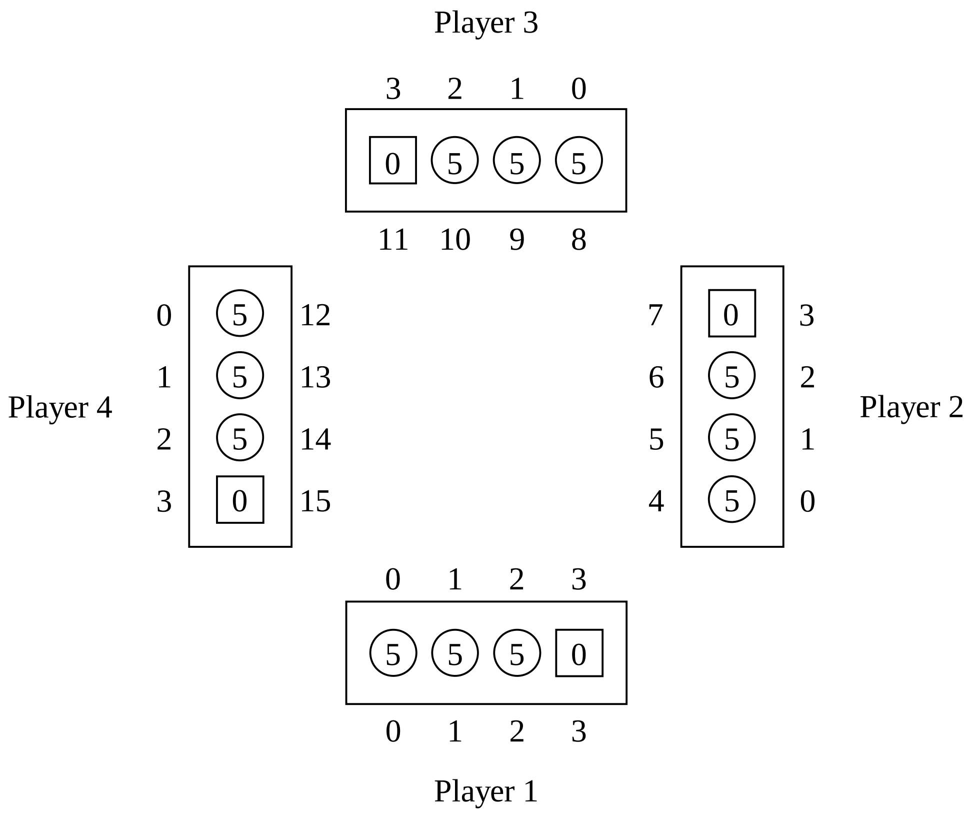
Tactic: Introduce an auxiliary mathematical function.

We name the desired mathematical function describing the initial state of all of the bowls "GameBoardInitialized". It is a function of the two parameters NP and NB. It must be true if all bowls are correctly initialized and false otherwise.

We introduce a mathematical function PlayerInitialized(p, NB) which will be true if the NB ordinary bowls and the special bowl in player p's collection of bowls are correctly initialized and false otherwise. GameBoardInitialized(NP, NB) will be true if and only if *every* player's bowls are correctly initialized, suggesting *universal quantification* (see the translator's glossary box above) over the players:

$$\text{GameBoardInitialized(NP, NB)} = (\wedge\ p : p \in \mathbf{Z} \wedge 1 \le p \le NP : \text{PlayerInitialized}(p, NB)\ )$$

In order to write an expression for PlayerInitialized(p, NB), we will need to refer to the individual bowls within each player's collection of bowls. Extending the diagram above to include such numbers, we obtain the following diagram. The new numbers are outside of the bowl areas.



Again, reread the English statement of the problem above and compare it with this diagram.

Strategy: Understand the meaning of the message.

Tactic: Express in an intermediate language (e.g. mathematically styled English).

From this diagram we can write the following draft of an expression for PlayerInitialized(p, 3):

> the number of stones in player p's bowl 0 = 5 $\wedge$
> the number of stones in player p's bowl 1 = 5 $\wedge$
> the number of stones in player p's bowl 2 = 5 $\wedge$
> the number of stones in player p's bowl 3 = 0

The number of stones in each bowl is represented by the value of that bowl's element of the array StonesInBowl. We need, therefore, a new auxiliary function that converts the new bowl number b within player p's collection to that bowl's index in the array StonesInBowl. We will call this function "GBIndex". This function will obviously depend on the player number p and the bowl number b. Presumably it will also depend on the game board parameter NB. It could also conceivably depend on NP, so we start with the parameter list (p, b, NB, NP) for this function. (It turns out below that GBIndex does not depend on NP, so we will drop that parameter later.)

Now we can write the following expression for PlayerInitialized(p, 3):

> StonesInBowl(GBIndex(p, 0, NB, NP)) = 5 $\wedge$ [NP will be dropped later]

StonesInBowl(GBIndex(p, 1, NB, NP)) = 5 $\wedge$
StonesInBowl(GBIndex(p, 2, NB, NP)) = 5 $\wedge$
StonesInBowl(GBIndex(p, 3, NB, NP)) = 0

or, for general NB,

PlayerInitialized(p, NB)

=

($\wedge$ b : b$\in$ **Z** $\wedge$ 0$\leq$b$\leq$NB-1 : StonesInBowl(GBIndex(p, b, NB, NP)) = 5)

$\wedge$ StonesInBowl(GBIndex(p, NB, NB, NP)) = 0

To define the function GBIndex, we refer to the diagram above and note that (1) the value of this function increases by NB+1 when p increases by 1 and (2) the value of this function increases by 1 when b increases by 1. An expression for the function GBIndex must, therefore, have the form p*(NB+1) + b + K, where K is an appropriate constant. Note that this expression depends on p, b and NB, but not on NP, so NP can be dropped from the list of parameters for the function GBIndex. K can be determined by considering the case p=1, b=0 for any NB (and any NP). Note that GBIndex(1, 0, NB) = 0 (see the diagram above):

GBIndex(1, 0, NB) = 0

=

1*(NB+1) + 0 + K = 0

=

K = -(NB+1)

= [GBIndex(p, b, NB) = p*(NB+1) + b + K, see above paragraph]

GBIndex(p, b, NB) = p*(NB+1) + b -(NB+1)

=

GBIndex(p, b, NB) = (p-1)*(NB+1) + b

Bringing together the various intermediate results above, the mathematical expression sought is:

GameBoardInitialized(NP, NB) = ($\wedge$ p : p$\in$ **Z** $\wedge$ 1$\leq$p$\leq$NP : PlayerInitialized(p, NB))

where PlayerInitialized(p, NB) =

($\wedge$ b : b$\in$ **Z** $\wedge$ 0$\leq$b$\leq$NB-1 : StonesInBowl(GBIndex(p, b, NB)) = 5)

$\wedge$ StonesInBowl(GBIndex(p, NB, NB)) = 0

and where GBIndex(p, b, NB) = (p-1)*(NB+1) + b

The above expressions for PlayerInitialized and GBIndex can be substituted for the function references to obtain a single, closed expression for GameBoardInitialized:

$$\begin{aligned}
&\quad \text{GameBoardInitialized(NP, NB)} \\
= \\
&\quad (\wedge\ p : p \in \mathbf{Z} \wedge 1 \leq p \leq NP : \quad (\wedge\ b : b \in \mathbf{Z} \wedge 0 \leq b \leq NB-1 : \text{StonesInBowl}((p-1)*(NB+1)+b) = 5) \\
&\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \text{StonesInBowl}(p*(NB+1)-1) = 0)
\end{aligned}$$

The next steps (see section 2.2 above) require that we reread and understand (interpret again) the English statement of the problem and our translation in mathematics (the mathematical expressions) and compare them. There are various ways of doing this, for example: (1) compare the final mathematical expressions against the English text, (2) draw a diagram from the mathematical expressions and compare it with the English text and our original diagram, (3) select a number of sample cases and calculate the number of stones in the selected bowls by reference to both the English and the mathematics texts independently, and verify that the results are the same (both correctly and incorrectly initialized board states should be included in the selection of sample cases), (4) examine the mathematical expressions for syntactical correctness, (5) examine the mathematical expressions for semantic plausibility, (6) examine the mathematical expressions for set compatibility (i.e. that the result of one function is consistent with the use of that result), etc. Drawing a hierarchical diagram of the composition of the functions in the mathematical expressions aids in the last point.

Persons experienced in interpreting English problem statements into mathematics would read the English text at the beginning of this section and would write the last expression above more or less directly, but even they would mentally go through most of the intermediate steps described above. Experience and practice does not enable one to skip intermediate steps in solving such problems, it only enables one to perform some of them mentally. Less experienced persons are well advised to write down all steps.

3.3.2 Initialization of a game board: a wrong "solution"

Common error: Confusing static and dynamic descriptions (confusing a mathematical expression and a program)

Often people accustomed to thinking in terms of the execution of a program will "solve" this problem in a way that reflects the structure of one possible program for initializing the game board. In effect, they confuse the desired mathematical expression describing the final state with a program for achieving that state. Such an erroneous approach is shown below.

Looking at the diagrams in section 3.3.1 above, we see that one fairly simple strategy for a program to initialize the board would be first to set all elements of the array StonesInBowl to 5 and afterward, set the elements of the array representing special bowls to 0. This thinking might lead one to write the following expression for GameBoardInitialized(NP, NB):

$$(\wedge\ i : i \in \mathbf{Z} \wedge 0 \leq i \leq NP*(NB+1)-1 : \text{StonesInBowl}(i) = 5) \qquad \text{[initialize all bowls]}$$

$$\wedge\ (\wedge\ p : p \in \mathbf{Z} \wedge 1 \leq p \leq NP : \text{StonesInBowl}(p*(NB+1)-1) = 0) \qquad \text{[initialize the special bowls]}$$

Stop, question and think: What is wrong with the above expression as a solution to the problem? Why does it not describe correctly the initial state of the game board?

What is wrong with this approach in general?

Answer these questions before proceeding.

In a correct initial state, every special bowl contains 0 stones and every ordinary bowl contains 5 stones. The first line in the above expression states that *all* bowls – ordinary and special – contain 5 stones each. The second line, which is anded with the first line, states that every special bowl contains 0 stones. Thus, the above expression states that every special bowl contains 5 stones *and* that every special bowl contains 0 stones. The two statements contradict each other, i.e. they cannot both be true. Expressed differently but equivalently, the above expression is always false for all positive NP and NB, even for a correctly initialized board, so the expression is not a solution to the problem. The above expression is an unnecessarily complicated way of writing the logical constant false.

Specific cases further illustrate this error. Consider, for example, the case NP=4, NB=3 (as in the diagrams in section 3.3.1 above), p=1 and b=3 (the first player's special bowl). The index for this bowl in the array StonesInBowl is GBIndex(p, b, NB) = (p-1)*(NB+1)+b = 3. Therefore, the number of stones in this bowl is StonesInBowl(3). The first line in the expression above contains the term StonesInBowl(3) = 5, and the second line contains the expression StonesInBowl(3) = 0. Thus, the entire expression is

$$\ldots \wedge \text{StonesInBowl}(3) = 5 \wedge \ldots \wedge \text{StonesInBowl}(3) = 0 \wedge \ldots$$

which is always false.

The person who wrote the above expression thought in terms of a two part program. The first part would initialize all bowls to 5. Afterwards, the second part would initialize only the special bowls to 0. The first line in the expression above describes the board state (the values of the elements of the array StonesInBowl) after the execution of the first part of the program but before the execution of the second part. The second line of the expression describes part of the state (the values of the elements of the array StonesInBowl representing the special bowls only) after the execution of the second part of the program. The first and second lines of the expressions are intended to describe different states, but the mathematical expression as written does not do this.

4. Inspecting and checking a mathematical expression

After composing a mathematical expression, one should check it for correctness. In the final analysis, one must ensure that it really means what it should or is intended to mean in terms of the task the expression describes or specifies. Methods and techniques for doing this have been outlined in the discussions about the various examples elsewhere in this monograph.

Before examining a draft expression for correctness, one can and should perform simpler checks. Many errors can be identified just by examining the expression or only small parts of it – without regard for the English text from which it has been translated and without regard for its intended meaning. In particular, the following categories of errors can be identified by an independent inspection and review of the expression and of its parts:

- syntactical errors (e.g. a+*b)
- type compatibility (e.g. "abc"<45.32, adding a non-numerical argument to anything)
- semantic plausibility (e.g. adding a variable counting apples to a variable representing tons of steel, adding a Canadian dollar amount to a Euro amount)

Inspecting an expression is often easier when one represents an expression hierarchically, i.e. in the form of a tree. Even when one does not write down an equivalent tree on paper, but constructs the tree only mentally, viewing an expression – and parts thereof – as a tree can be very helpful in identifying some types of errors. Representing an expression as a tree is, therefore, also discussed in this chapter, see section 4.3 below.

4.1 Syntax

A mathematical expression in the context of this monograph represents a function (and its intended evaluation). Functions are most frequently represented in one of three notational forms:

- classical functional notation, in which the function name is followed by a list of arguments separated by commas and enclosed in parentheses, e.g. sum(x, y), or(a, b)
- infix notation, in which the function name or symbol is preceded by one argument and followed by a second argument, e.g. x+y, $a \vee b$
- quantified notation, in which the function, one or more "bound" variable(s), the range of the bound variable(s) and the arguments of the function are given in any one of several formats, e.g. $(+\ i : i \in \mathbf{Z} \wedge 1 \le i \le n : i*(i+1))$, $(\wedge\ i : i \in \mathbf{Z1} : A(i) \wedge B(i))$, $\forall i \in \mathbf{Z1}(A(i) \wedge B(i))$, $(\forall i,j | 1 \le i,j \le n : C(i) \wedge D(j))$. The range of the bound variable(s) and each argument are given in the form of an expression. See Appendix C below for further information on notational forms for quantified expressions.
- series notation for quantified expressions, e.g. $\sum_{i=1}^{n} i*(i+1)$, $\mathbf{and}_{i=1}^{n} A(i) \wedge B(i)$.

An expression may also consist of a variable name (e.g. x, y(i)) or of a constant (e.g. 4, "abc") by itself. Any expression may be enclosed in parentheses where desired.

Functions are *composed* by writing an appropriate expression for an argument, i.e. by allowing one expression to be a component of another, e.g.

- product(sum(x, y), z)
- product(x+y, z)
- sum(x, y)*z

- $(x+y)*z$
- and(a, b)
- or(c, d)
- and($c \lor d$, e)
- $(c \lor d) \land e$

A syntactically valid expression may be built up by applying the above rules only. A sequence of symbols that cannot be formed by applying the above rules is not a syntactically valid expression.

4.2 Type compatibility

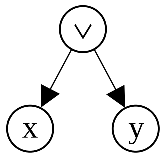
Every function is defined in such a way that the values of its arguments must be elements of certain sets characteristic of the particular function. A value of an argument violating this requirement is incompatible with its usage. For example, the expression $4 \land 6$ is syntactically valid (4 is an expression, 6 is an expression, and $\land$ is the infix symbol for a function), but the and function ($\land$) is not defined for numerical arguments. The expression $4 \land 6$ is, therefore, not meaningful. It is clearly a mistake.

4.3 Tree representation

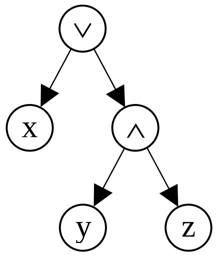
An expression can be represented by a tree. The root and each internal node in the tree represents an operation (function). Each branch connects an operation with one of its arguments. Each leaf (end node) represents a variable or constant.

When an expression is represented as a tree, it is easier to see some of the relationships between parts of the expression. In particular, type compatibility is easy to check visually with a tree, especially with long expressions. When the sequence of symbols constituting the expression is long, related parts may be far apart and the connection not evident, but in the tree, an arrow connects the directly related parts of the expression.

Example: The expression $x \lor y$ is represented by the following tree.



Example: The expression $(x \lor (y \land z))$ is represented by the following tree.

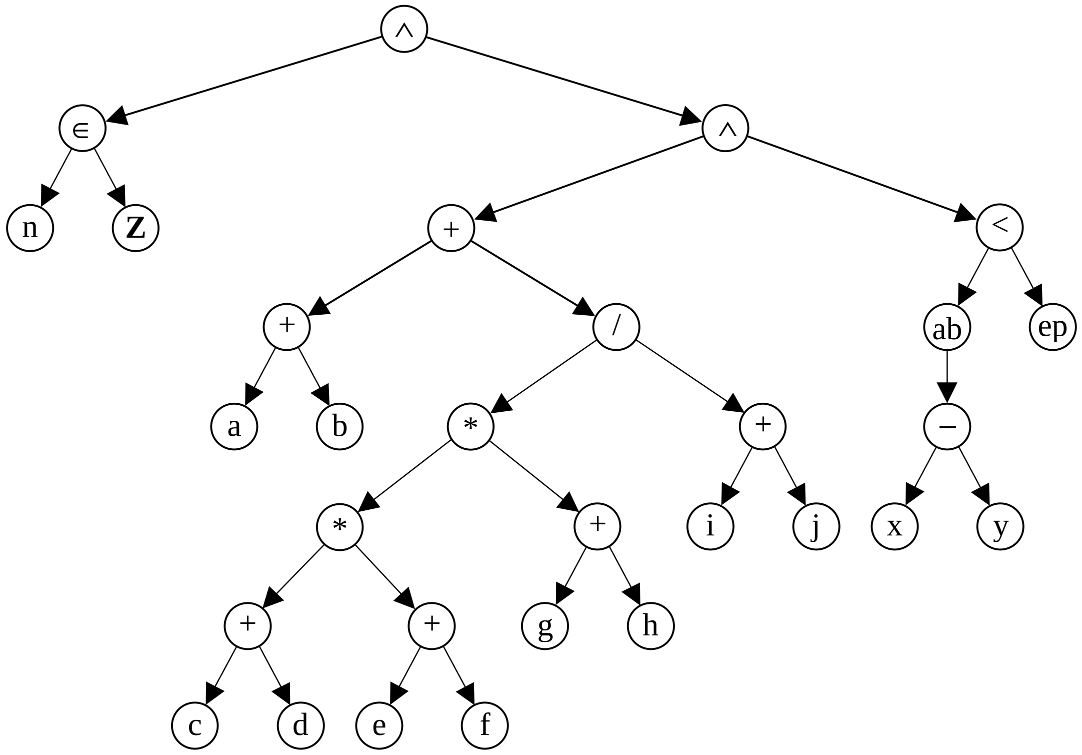


Notice that the second argument of the $\lor$ operation is a subexpression, represented by a subtree.

Example: Consider the expression

$n \in \mathbf{Z} \wedge (((a+b)+(c+d)*(e+f)*(g+h)/(i+j)) \wedge (ab(x-y)<ep))$

Many people will not immediately see the error in the above expression because of the length of the expression and because of the distance between the symbols for the two incompatible operators. The error is easier to see when one draws the tree corresponding to the above expression:



While drawing this tree one will notice that one of the arguments of the ∧ operation is the result of an addition. The ∧ operation requires a Boolean value for each of its arguments, but the result of the addition is a number. I.e., the number is incompatible with its usage. This is a type incompatibility as described in section 4.2 above. The expression is, therefore, wrong.

Quantified expressions can also be represented as a tree. At least two views of a quantified expression are valid. The corresponding trees are similar, but not the same.

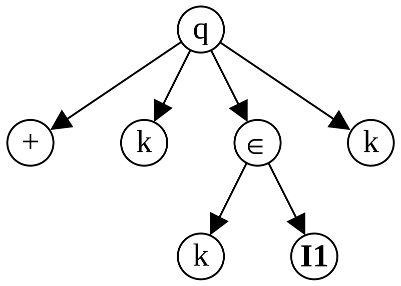
In the first view, the quantification itself is considered to be the main operation. Its four arguments are

- the operation to be repeated,
- the dummy variable (also called the quantified variable or the bound variable),
- an expression defining the range of the dummy variable and
- the expression being quantified, i.e. the expression which represents the arguments of the operation to be repeated.

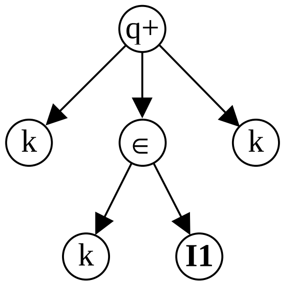
The expression defining the range of the dummy variable must be a Boolean expression, i.e. its values must be false or true. This expression must define values of the dummy variable that are consistent with the usage of the dummy variable in the expression being quantified. The values of the expression being quantified must be consistent (compatible) with the operation.

In the second view, the operation over which quantification is applied is considered to be the main operation, which is represented by the root node of the entire quantified expression. Then the arguments of the quantified operation are the last three listed above, i.e. the dummy variable, the range and the expression being quantified.

Example: Consider the quantified expression (+ k : k∈**I1** : k). The first view described above of this expression (the quantification itself is the main operation) is represented by the tree below.



The second view (the quantified operation is the main operation) is represented by the tree below. In such a tree one should represent the quantified operation (here addition) in a different way than the ordinary, unquantified operation because the quantified operation and the ordinary operation have different arguments. The quantified addition has three arguments: (1) the dummy variable, (2) a Boolean expression defining the range of the dummy variable and (3) the numerical expression being quantified, whereas ordinary addition has two arguments, both numerical.



Either of the two above forms may be used to represent a quantified expression or subexpression in a tree.

4.4 Semantic plausibility

“You can’t add apples and oranges.” This old saying observes that certain operations on unlike quantities are meaningless. Comparable statements expressed mathematically are, therefore, invalid.

Typical examples are performing numerical calculations on numbers not appropriately related or dimensionally different, e.g. multiplying a temperature by a velocity, comparing a person’s name with an address, adding a temperature in Celsius and a temperature in Fahrenheit, etc. In such cases the subexpressions in question are almost certainly wrong.

Similar situations arise that are suspicious, but not necessarily wrong. If, for example, in an expression an integer is compared with or added to a real number, the expression should be checked carefully for validity. Mathematically there is nothing wrong in such cases, but it is questionable whether an actual task description would give rise to such a mathematical formulation.

5. Conclusion

In this monograph the nature of mathematics as a language and the process of translating from English (or any other natural language) into mathematics has been examined. Fluency both in English and in mathematics is a prerequisite for successful translation, but it is not enough. The translator must also be familiar with the process of translating as such and with appropriate techniques for translating. Finally, but not less important, the translator must also be familiar with the field (application area) to which the material to be translated relates.

Among the techniques of use to the translator are the strategies and tactics developed in chapter 3 above and summarized in Appendix B below. Translators will also refer often to the glossary in Appendix A below of frequently occurring English terms with the corresponding mathematical objects and symbols. The reader should add to both of these collections as they gain experience in translating from English into mathematics.

The linguistic nature of the process of formulating word problems mathematically appears to be completely neglected in books on mathematics. This monograph is intended to fill this important gap.

This monograph is an early draft. Reader's comments, suggestions and contributions, especially examples, are solicited. Please send them by email to the author at Baber@McMaster.CA.

Appendix A. English to mathematics translator's glossary

English	mathematics
and, but	and, $\wedge$
equal, equals	=
exchange, rearrange, different order (sequence), merge, copy, sort	permutation
(for) all, any, each, every	$\forall$, $\wedge$ (and) series, for all, universal quantification
(for) no, none	$\forall$, $\wedge$ (and) series, for all, universal quantification with negated assertion
if, when, whenever ... then ...	logical implication, ... $\Rightarrow$ ...
in	$\in$
integer	$...\in \mathbf{Z}$
or	or, $\vee$
other, unequal	$\neq$
same, search, find, equal, present	=
sorted [$A(1) \leq A(2) \leq ... \leq A(n)$]	$(\wedge\ i : i\in \mathbf{Z} \wedge 1\leq i\leq n\text{-}1 : A(i)\leq A(i+1)\ )$ $\text{and}_{i=1}^{n-1}\ A(i)\leq A(i+1)$
there is (are), there exist(s), (for) some, at least one	$\exists$, $\vee$ (or) series, existential quantification

Appendix B. Strategies and tactics

Strategy	tactics
Close the gap between the English text and mathematics.	Express implicit information explicitly. Reduce vagueness and ambiguity. Reword the English text to bring it closer to mathematics.
Divide and conquer (complexity).	Construct a table. Distinguish between specific cases. Introduce an auxiliary mathematical function. Modularize.
Obtain all needed information.	Ask the author of the task description. Identify gaps in the description of the task. Identify implicit “information”. Identify the data present and ask questions about related details. Question explicitly whether implied information may be assumed. Question whether missing information is really needed. Read carefully, thoroughly and precisely.

Understand the meaning of a message.	Describe a specific instance of the general problem. Distinguish between essential objects and background information. Draw a diagram. Express in an intermediate language (e.g. drawing, mathematically styled English, picture). Identify all objects referred to in the message. Identify implicit – but not necessarily true – information. Identify missing information. Identify relationships between essential objects referred to in the message. Identify special cases, especially ones not explicitly mentioned.

Appendix C. Notation: quantification

Universal and existential quantification ("for all" and "there exists", $\forall$ and $\exists$ respectively) is common in mathematical logic and such expressions occur often in software engineering. Also addition is frequently quantified in all engineering fields. In software engineering, still other operators are frequently quantified in the same way. Although different notational forms are often used for these different purposes, they all have a common structure: One or more dummy variables are introduced for the quantification, a range of values for the dummy variables is defined, and the operation is applied repeatedly to values generated by the quantified expression (function) applied to the dummy variables. Thus, all forms of such quantification can be viewed as a function with four arguments: (1) an operation, (2) the names of dummy variables, (3) a Boolean expression defining the ranges of the values of the dummy variables and (4) an expression (function of the dummy variables) to whose values the operation is to be repeatedly applied.

Several different notational forms are used in mathematics for universal and existential quantification. For other operations, e.g. addition and multiplication, the series notation (e.g. $\sum_{i=1}^{n} f(i)$, $\prod_{i=1}^{n} g(i)$) is often used.

Set formation in mathematics is another form of quantification that is often written in a still different way, with the set union operation ($\cup$) implied, e.g. $\{x \mid x \in \mathbf{Z} \wedge 1<x<n\}$. In some cases this last notational form can be ambiguous, e.g. it is sometimes not clear whether a particular variable is a dummy variable or a free variable. Sometimes the dummy variable is not given explicitly, but only implied, as in the expression $\{(2*x+3) \mid x \in \mathbf{Z} \wedge 1<x<n\}$.

Because all of these notational forms refer to the same mathematical structure, it is convenient to use a common notational form for all. Such a notational form that has come into common use in some subareas of software engineering is introduced below and is used in this monograph.

The notation (**op** i : r(i) : exp(i)) means

exp(i1) **op** exp(i2) **op** exp(i3) ...

where **op** is any operator (function) satisfying the commutative and associative laws, where r is a Boolean function, and where i1, i2, i3, etc. are all the values of i for which r(i) is true. The expression (function) exp need not be Boolean; its range may be any set consistent with the definition of the particular **op**.

Among the many operators frequently appearing in such quantified expressions are $\wedge$, $\forall$, $\vee$, $\exists$, +, $*$, & (concatenation of sequences), $\cap$, $\cup$, min, max. For typographical reasons synonyms are also seen (e.g. A for $\wedge$ and $\forall$, E for $\vee$ and $\exists$).

The above definition can be extended to cover situations in which **op** is not commutative or not associative (or both). If **op** is not commutative, then the order in which the elements i1, i2, etc. appear must be defined. Usually, this order will be the order defined on the set to which the values of i1, i2, etc. belong. This set will often be specified within the Boolean function r. If **op** is not associative, the grouping (implied parentheses) must be specified, e.g. left to right.

If quantification over the empty set is specified (i.e. if there is no value for i for which r(i) is true), then the value of the entire expression is defined to be the identity element of the operator **op**. (Why is this convention appropriate and convenient?)

The above definitions apply to quantification over a finite set (i.e. the values i satisfying r(i) are finite in number). In software engineering, quantification over finite sets is typical, so these definitions are normally sufficient. When the quantification is over an infinite set, the meaning of such quantified expressions must be suitably defined, usually as the limit of a sequence of quantifications over finite sets. The specific definition depends on the operator **op**. One should keep in mind that such quantification over infinite sets is not always defined. Quantification of addition over an infinite set in which both positive and negative terms to be added is a classical example.

$(\wedge\ i : i\in S : exp(i))$, $(A\ i : i\in S : exp(i))$ and $(\forall\ i : i\in S : exp(i))$ are often written instead of $(\forall i\in S{:}exp(i))$ or similar forms. Similarly, $(\vee\ i : i\in S : exp(i))$, $(E\ i : i\in S : exp(i))$ and $(\exists\ i : i\in S : exp(i))$ are often written for $(\exists i\in S{:}exp(i))$ or similar forms.

Many target groups of readers, e.g. clients of software engineers, are not particularly familiar with mathematical symbols such as $\wedge$, $\vee$, $\forall$, $\exists$, etc. Many of them find the resulting, often rather dense, expressions difficult to read and understand. Readability can often be facilitated by using more descriptive notation such as **and**, **or**, etc. instead of the mathematical symbols. Splitting an expression over two or more lines and indenting appropriately can often improve readability of long expressions considerably.

For readers in the traditional engineering fields, the series notation will often be the most readable form. The series notation can be generalized to commutative and associative operations (functions) other than addition in the obvious manner, e.g.

$$\mathbf{and}_{i=1}^{n}\ exp(i)$$

etc. Formally, $\mathbf{op}_{i=1}^{n}\ exp(i)$ means $(\mathbf{op}\ i : i\in \mathbf{Z} \wedge 1\le i\le n : exp(i))$.

Appendix D. Notation: derivations and proofs

Derivations and proofs are often written in the following format:

```
        expression 1
=                                                        [comment]
        expression 2
=
        expression 3
```

etc. This is defined to mean (expression 1 = expression 2) and (expression 2 = expression 3), etc. On lines beginning with the = symbol, only the = symbol is mathematically significant; anything appearing on the same line after the = symbol (e.g. "[comment]" above) is a comment, typically justifying the step in the derivation or the proof.

Similarly,

```
        expression 1
⇒                                                        [comment]
        expression 2
⇒
        expression 3
```

is defined to mean (expression 1 $\Rightarrow$ expression 2) and (expression 2 $\Rightarrow$ expression 3).

Appendix E. Undetermined (undefined) values of expressions

Sometimes the value of an expression or subexpression is undetermined or undefined. A common example is an expression containing a reference to an array element whose index value is out of range, e.g. A(n+1) when the array is defined (declared) for index values ranging from 1 to n (or 0 to n) only.

When an expression can contain a reference to a variable or a function (subexpression) whose value is not defined or cannot be determined, one must define mathematically how such occurrences should be handled and interpreted. Some of the possibilities are described in Baber, The Spine of Software, section A0.3.2, pages 271-276, Bijlsma and Parnas (both references).

The functions represented by such undefined expressions or parts thereof can be viewed in mathematical terms as *partial functions*, i.e. functions whose values are not defined for all possible values of their arguments. In order to handle such undefined expressions in well defined ways, one can extend the definitions of the partial functions in question to make them total functions, i.e. functions whose values are defined for all possible values of their arguments.

One way of turning a Boolean expression representing a partial function into a total function that makes the desired meaning of an expression E explicit is as follows.

1. Introduce a new, unique element "*", which is not a valid value for any argument of any function represented by the expression in question or by any of its subexpressions – with one exception: the function ≡, see point 3 below.

2. For every subexpression S (also the entire expression E) representing a partial function Fs, extend Fs to a total function Fsx in the following way: Define the value of Fsx to be the same as Fs for all argument values for which Fs is defined and "*" otherwise. Note that this implies that if "*" is an argument of any such extended function Fsx, the function value is also "*", i.e. Fsx propagates the value "*".

   Also the function = (normal equality) will normally be viewed in this way as a partial function, no argument of which may be undefined or "*". In this case, it, too, must be extended in this manner. (Alternatively, the function = may, if appropriate in particular situations, be viewed as a total function as the function ≡ in point 3 below.)

3. Replace, as needed to achieve the desired effect, Boolean subexpressions B by expressions involving (i) B or a component of B, (ii) the (total) function ≡ (see the table below), and (iii) the constants true, false or "*" as needed to achieve the meaning desired when one or more arguments are not in the defined domains or are "*". Most commonly, one will replace a subexpression of the form ¬B by (B≡false) and a subexpression of the form B by (B≡true). This function ≡ is defined originally as a total function on the set {false, "*", true}×{false, "*", true} to the set {false, true}, so it does not need to be extended as described in point 2 above. With this definition, if the value of the expression B is undefined ("*"), the expressions (B≡true) and (B≡false) will both be false. The function ≡ is defined by the following truth table:

x	y	x≡y
false	false	true
false	“*”	false
false	true	false
“*”	false	false
“*”	“*”	true
“*”	true	false
true	false	false
true	“*”	false
true	true	true

The above way of interpreting expressions when the values of arguments can be undetermined, undefined or outside the domains of the functions in question is useful, but it does not cover every need. Neither does any of the other methods described in the references above. There is no universally correct way to handle undefined expressions and parts thereof. When undefined arguments can arise and are of possible consequence, the software engineer must give careful thought to what meaning of the expression is desired or expected in such cases and must select an appropriate convention among the many possibilities.

Some of the ways of handling expressions with undetermined or undefined values fulfill all the laws of Boolean algebra, while others do not. The former, while convenient, do not satisfy all needs arising in specific situations and do not, therefore, represent universal solutions to the problem of undetermined or undefined values of variables and expressions.

Example 1: Consider the expression

$$n \in \mathbf{Z} \wedge i \in \mathbf{Z} \wedge 0 \leq i < n \wedge X(n)=y$$

where the array X is defined (declared) only for index values from 0 to n-1 inclusive. X(n) is not defined (declared), so we would like the entire expression to be evaluated to false, not left undefined. We can turn this into an expression representing a total function by applying the total function ≡:

$$[n \in \mathbf{Z} \wedge i \in \mathbf{Z} \wedge 0 \leq i < n \wedge X(n)=y] \equiv \text{true}$$

If it is known that values are defined for the variables n and i (but not necessarily y), the same effect can be achieved by applying the total function ≡ at a lower level:

$$n \in \mathbf{Z} \wedge i \in \mathbf{Z} \wedge 0 \leq i < n \wedge [(X(n)=y) \equiv \text{true}]$$

Following the convention that = is a partial function, then if the value of either X(n) or y (or both) is undefined (not declared), the value of the = function will be “*”, which is not equivalent to true, so the entire expression will evaluate to false.

Example 2: Consider the expression [Parnas]

$x \in \mathbf{R} \wedge (x \geq 0 \wedge y = \sqrt{x} \vee x < 0 \wedge y = \sqrt{-x})$

Interpreted strictly, the value of this expression will be true if x=0 and it will be undefined otherwise. If x is positive, then the square root of –x will be undefined, which will result in the values of the higher level subexpressions and the expression as a whole to be undefined. Similarly, if x is negative, then the square root of x will be undefined and the value of the entire expression will be undefined.

The desired interpretation of this expression is probably that it should be true whenever y is the square root of the absolute value of x. The simplest way to rewrite the expression would be to write $y = \sqrt{|x|}$. As an exercise, however, we wish to modify the above expression to reflect the desired meaning. Applying the function ≡ at the highest level would result in an expression equivalent to x=0, not the desired result. By applying the function ≡ to the = functions, we write

$x \in \mathbf{R} \wedge (x \geq 0 \wedge [(y = \sqrt{x}) \equiv \text{true}] \vee x < 0 \wedge [(y = \sqrt{-x}) \equiv \text{true}])$

This value of this expression will be undefined (“*”) only if either x<0 or x≥0 is undefined (“*”) (or both are undefined), i.e. if x is not a real number. The expression can be extended to a total function by applying the function ≡ to the right subexpression of the highest level function ∧:

$x \in \mathbf{R} \wedge \mathbf{(}x \geq 0 \wedge [(y = \sqrt{x}) \equiv \text{true}] \vee x < 0 \wedge [(y = \sqrt{-x}) \equiv \text{true}]\mathbf{)} \equiv \text{true}$

or by applying the function ≡ to the entire expression:

$\mathbf{(}x \in \mathbf{R} \wedge x \geq 0 \wedge [(y = \sqrt{x}) \equiv \text{true}] \vee x < 0 \wedge [(y = \sqrt{-x}) \equiv \text{true}]\mathbf{)} \equiv \text{true}$

Example 3: Care must be exercised when a Boolean subexpression is negated. Consider, for example, the expression ¬B where the subexpression B is not defined. Strictly, the expression ¬B is, therefore, also undefined. Most often we are interested in distinguishing between true or otherwise, i.e. we want an undefined expression (here ¬B) to default to false. Only very seldom do we want an undefined expression to default to true.

If we apply the function ≡ at the lowest level, we convert the above expression to

$\neg(B \equiv \text{true})$

which, when B is “*”, evaluates to

$\neg(\text{“*”} \equiv \text{true})$

=

$\neg(\text{false})$

=

true

which is not the desired value.

Usually, an expression of the form $\neg B$ should be converted to $(B \equiv \text{false})$, which will evaluate to false when the value of B is either undefined (“*”) or true.

Initially, the reader should first translate a task from English to mathematics without regard to possibly undefined arguments or subexpressions and only when the mathematical expression is complete, checked and correct for valid arguments, consider undefined or invalid arguments and modify the expression as outlined in this appendix. Only after gaining sufficient experience should one consider undefined and invalid arguments from the very beginning. I.e., one should divide the learning process in order to conquer it.

In summary, the software engineer writing an expression whose value may be undetermined or undefined in some situations should rewrite the expression in such a way that it evaluates to the desired values in all cases, including those in which some variables or intermediate values may be outside the corresponding domains or even undefined.

Appendix F. Bibliography

* Baber, Robert L., *The Spine of Software: Designing Provably Correct Software — Theory and Practice*, John Wiley & Sons, Chichester, 1987.

* Baber, Robert L., *Error-free Software: Know-How and Know-Why of Program Correctness*, John Wiley & Sons, Chichester, 1991.

Baber, Robert L., Slides for the course “Software Design II”, http://www.cas.mcmaster.ca/~baber/Courses/2B03/Slides07EnglMath.pdf. See also other files in the same web directory. 2002.

Bijlsma, A., “Semantics of Quasi-Boolean Expressions”, chapter 3, pp. 27-35, in Feijen, W. H. J. et al., *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, Springer-Verlag, New York, 1990.

Dean, Neville, *The Essence of Discrete Mathematics*, Prentice Hall, Harlow, England, 1997.

Denvir, Tim, *Introduction to Discrete Mathematics for Software Engineering*, Macmillan Education Ltd., Basingstoke, 1986.

Gries, David and Schneider, Fred B., *A Logical Approach to Discrete Math*, Springer-Verlag, New York, 1993.

Hart, John, *Consider a Spherical Cow: A Course in Environmental Problem Solving*, University Science Books, Sausalito, California, 1988.

Hart, John, *Consider a Cylindrical Cow: More Adventures in Environmental Problem Solving*, University Science Books, Sausalito, California, 2001.

Ince, D. C., *An Introduction to Discrete Mathematics, Formal System Specification, and Z*, Clarendon Press, Oxford, 1992.

LeBlanc, Jill, *Thinking Clearly: A Guide to Critical Reasoning*, W. W. Norton & Company, New York, 1998.

Lipschutz, Seymour and Lipson, Marc Lars, *2000 Solved Problems in Discrete Mathematics*, Schaum’s Solved Problems Series, McGraw-Hill, New York, 1992.

Parnas, David Lorge, “Predicate Logic for Software Engineering”, *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, pp. 856-862, 1993 September.

Parnas, David L., “Predicate Logic for Software Engineering”, Chapter 3, pp. 49-65, in Hoffmann, Daniel M. and Weiss, David M. (eds.), *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, Boston, 2001.

* The books marked * above are available online in pdf files without charge. See web page http://www.cas.mcmaster.ca/~baber/Books/Books.html for further information on each of these books and to download the files containing them.

Additional exercises relating to software engineering can be found in

- several books listed above, especially *The Spine of Software* and *Error-free Software*,
- the file http://www.cas.mcmaster.ca/~baber/Courses/46L03/MRSDExer.pdf and
- other files with names beginning with “MRSDExer” in the same directory.