Appendix B

Pseudocode

Pseudocode is commonly used in mathematics and computer science to present algorithms. In this appendix, we will describe the pseudocode used in the text. If you have studied a programming language, you may see a similarity between our pseudocode and the language you studied. This is no accident, since programming languages are also designed to express algorithms. The syntax, or "grammatical rules," of our pseudocode will not be as rigid as that of a programming language since we do not require that it run on a computer. However, pseudocode serves much the same purpose as a programming language.

As indicated in the text, an algorithm is a specific set of instructions for performing a particular calculation with numerical or symbolic information. Algorithms have *inputs* (the information the algorithm will work with) and *outputs* (the information that the algorithm produces). At each step of an algorithm, the next operation to be performed must be completely determined by the current state of the algorithm. Finally, an algorithm must always terminate after a finite number of steps.

Whereas a simple algorithm may consist of a sequence of instructions to be performed one after the other, most algorithms also use the following special structures:

- Repetition structures, which allow a sequence of instructions to be repeated. These structures are also known as *loops*. The decision whether to repeat a group of instructions can be made in several ways, and our pseudocode includes different types of repetition structures adapted to different circumstances.
- Branching structures, which allow the possibility of performing different sequences of instructions under different circumstances that may arise as the algorithm is executed.

These structures, as well as the rest of the pseudocode, will be described in more detail in the following sections.

§1 Inputs, Outputs, Variables, and Constants

We always specify the inputs and outputs of our algorithms on two lines before the start of the algorithm proper. The inputs and outputs are given by symbolic names in

usual mathematical notation. Sometimes, we do not identify what *type* of information is represented by the inputs and outputs. In this case, their meaning should be clear from the context of the discussion preceding the algorithm. Variables (information stored for use during execution of the algorithm) are also identified by symbolic names. We freely introduce new variables in the course of an algorithm. Their types are determined by the context. For example, if a new variable called a appears in an instruction, and we set a equal to a polynomial, then a should be treated as a polynomial from that point on. Numerical constants are specified in usual mathematical notation. The two words *true* and *false* are used to represent the two possible truth values of an assertion.

§2 Assignment Statements

Since our algorithms are designed to describe mathematical operations, by far the most common type of instruction is the *assignment* instruction. The syntax is

<variable> := <expression>.

The symbol := is the same as the assignment operator in Pascal. The meaning of this instruction is as follows. First, we *evaluate* the expression of the right of the assignment operator, using the currently stored values for any variables that appear. Then the result is stored in the variable on the left-hand side. If there was a previously stored value in the variable on the left-hand side, the assignment *erases* it and *replaces* it with the computed value from the right-hand side. For example, if a variable called i has the numerical value 3, and we execute the instruction

$$i := i + 1,$$

the value $3 + 1 = 4$ is computed and stored in i. After the instruction is executed, i will contain the value 4.

§3 Looping Structures

Three different types of repetition structures are used in the algorithms given in the text. They are similar to the ones used in many languages. The most general and most frequently used repetition structure in our algorithms is the WHILE structure. The syntax is

WHILE <condition> DO <action>.

Here, <action> is a sequence of instructions. In a WHILE structure, the action is the group of statements to be repeated. We always indent this sequence of instructions. The *end* of the action is signalled by a return to the level of indentation used for the WHILE statement itself.

The <condition> after the WHILE is an assertion about the values of variables, etc., that is either true or false at each step of the algorithm. For instance, the condition

$$i \leq s \text{ AND divisionoccurred } = \text{ false}$$

appears in a WHILE loop in the division algorithm from Chapter 2, §3.

When we reach a WHILE structure in the execution of an algorithm, we determine whether the condition is true or false. If it is *true*, then the action is performed once, and we go back and test the condition again. If it is still true, we repeat the action once again. Continuing in the same way, the action will be repeated as long as the condition remains true. When the condition becomes false (at some point during the execution of the action), that iteration of the action will be completed, and then the loop will terminate. To summarize, in a WHILE loop, the condition is tested *before* each repetition, and that condition must be *true* for the repetition to go on.

A second repetition structure that we use on occasion is the REPEAT structure. A REPEAT loop has the syntax

$$\text{REPEAT} < \text{action} > \text{ UNTIL } < \text{condition} >.$$

Reading this as an English sentence indicates its meaning. Unlike the condition in a WHILE, the condition in a REPEAT loop tells us when to *stop.* In other words, the action will be repeated as long as the condition is *false.* In addition, the action of a REPEAT loop is always performed at least once since we only test the condition *after* doing the sequence of instructions representing the action. As with a WHILE structure, the instructions in the action are indented.

The final repetition structure that we use is the FOR structure. We use the syntax

FOR each s in S DO <action>

to represent the instruction: "perform the indicated action for each element $s \in S$." Here S is a finite set of objects and the action to be performed will usually depend on which s we are considering. The order in which the elements of S are considered is not important. Unlike the previous repetition structures, the FOR structure will necessarily cause the action to be performed a fixed number of times (namely, the number of elements in S).

§4 Branching Structures

We use only one type of branching structure, which is general enough for our purposes. The syntax is

$$\text{IF} < \text{condition} > \text{ THEN } < \text{action1} > \text{ ELSE } < \text{action2} >.$$

The meaning is as follows. If the condition is true at the time the IF is reached, action 1 is performed (once only). Otherwise (that is, if the condition was false), action2 is performed (again, once only). The instructions in action 1 and action2 are indented, and

the ELSE separates the two sequences of instructions. The end of action2 is signalled by a return to the level of indentation used for the IF and ELSE statements.

In this branching structure, the truth or falsity of the condition selects which action to perform. In some cases, we *omit* the ELSE and action2. This form is equivalent to

IF $<$condition$>$ THEN $<$action1$>$ ELSE do nothing.