An Informal Language for Writing Algorithms

An algorithm is a set of instructions that can be followed in order to solve a particular problem. It can be thought of as no more then an outline or step by step solution to the problem you are trying to solve. To be an algorithm, the set of instructions must be unambiguous. Algorithms are used in every day life. For example, a recipe for baking a cake is an algorithm and so is a set of directions for finding your home. Sometimes programmers give a narrow definition, defining an algorithm to be a series of instructions written with sufficient detail that they can easily be translated into a computer language. We’ll assume this definition for the rest of this handout.

Writing algorithms is an important part of the programming process, which involves the steps below:

1) Analyzing the problem.
2) Constructing an algorithm to solve the problem.
3) Translating the algorithm into a programming language.
4) Testing and debugging the program.

It is a well known fact of programming life that if an algorithm construction is not done well, a programmer will spend excessive amounts of time testing and debugging.

Algorithms are written in languages ranging from natural languages like English to programming languages such as C++. However, natural languages tend to be ambiguous and difficult to translate into a programming language. So why not just write algorithms using a programming language such as C++ or Pascal? The answer is that such languages are rigidly defined, and trying to use the proper syntax tends to distract one from concentrating on the algorithm. Computer programmers usually use an informal language called “pseudocode”, which is particularly well suited to express algorithms. A key idea when using pseudocode is that one need not be concerned about syntax (spelling, punctuation, etc.), so one can focus on the algorithm. Once the algorithm has been written in pseudocode, it is normally a routine matter to translate the pseudocode into a programming language.

There are many dialects of pseudocode, but most are similar to the one described below, which is powerful, concise, and easy to translate into C++ and many other programming languages. The language requires only a small number of statements. It is based on the fact that when writing an algorithm, most actions can be built from the following:

1) Assigning a value to a variable (this can involve using operations such as +, -, <, etc.)
2) Reading (inputting) a value and storing it in a variable.
3) Writing (displaying) a number, string, or a value stored in a variable.
4) Performing one or more of the above statements conditionally, i.e. if a certain condition is true.
5) Performing one or more of the above statements repeatedly, while a certain condition is true.

It is clear that the notion of a variable is important, so let’s look at it more carefully. A variable, also called an identifier, is a named memory cell that can hold a single value. Often we will use a diagram such as the one below to represent a variable and its value.

\[
\begin{array}{c}
N \\
12
\end{array}
\]

In this case, the name of the variable is “N” and the value of the variable is 12. Variables will be used to “remember” values in case they are needed later in the algorithm. For example, if we need to remember the value 7, we can simply store it in a variable.

Variable names can be as long as desired, but should begin with an alphabetic character. Well chosen variable names can make an algorithm much easier to read, you should invest the time needed to choose names that will help the reader. When possible, use names that have mnemonic value, i.e. that help the reader understand the intended meaning of the variable, constant or subprogram or function.

Variables can hold various types of data, for example, they can hold integers such as 1 or -6. They can also hold real numbers such as 1.2 or 3.14. Finally variables can contain strings such as “cat” or “hello”.
We’ll soon look carefully at each of the statements in our informal language, but first let’s get a quick preview by looking at a few examples.

**Algorithm 1:** An algorithm that reads a number input by a user, then displays a message saying whether the number is odd or even. Note that the short name, “N”, is acceptable since the number has no special properties.

```plaintext
write "Enter a number"
read N

if (N mod 2 == 0) then  // if number is divided by 2 and remainder is 0
  write N, " is even"
else
  write N, " is odd"
end if
```

**Algorithm 2:** Similar to algorithm 1), but the user is given the option to repeat the process.

```plaintext
write "This algorithm determines whether numbers are odd or even."
write "To end this algorithm, enter a negative or zero."
write "Enter the first number to check: "
read N

while (N>0) do
  if (N mod 2 == 0) then  // if number is divided by 2 and remainder is 0
    write N, " is even"
  else
    write N, " is odd"
  end if
  write "Enter another number: "
  read N
end while
```

**Algorithm 3:** An algorithm that allows the user to enter a series of numbers. Input ends when zero is entered, at which time the sum of all the numbers is displayed.

```plaintext
Sum = 0
write "Enter the numbers you want to add – Enter 0 to stop"
write "Enter the first number ==> "
read N

while (N!=0) do
  Sum = Sum + N
  write "Enter the next number ==> "
  read N
end while

write "The sum of the number is ", Sum
```

Before we describe carefully the statements in our informal language, we’ll review some needed ideas.
Arithmetic Operations

Many algorithms require arithmetic computations and we’ll be using the following operations.

+ (addition)  
- (subtraction)  
* (multiplication)  
/ (division)

An operator that is surprisingly useful is the \texttt{div} operator. It is helpful when working with integers (whole numbers). By definition, \texttt{A div B} is the integer result, or quotient, obtained when \texttt{A} is divided by \texttt{B}. For example,  

\[
11 \text{ div } 3 = 3 \\
12 \text{ div } 3 = 4 \\
3 \text{ div } 12 = 0
\]

The \texttt{mod} operation is also useful when working with integers. By definition, \texttt{A mod B} is the remainder when integer \texttt{A} is divided by integer \texttt{B}. For example,  

\[
11 \text{ mod } 3 = 2 \\
12 \text{ mod } 3 = 0 \\
3 \text{ mod } 12 = 3
\]

The “\texttt{inc}” and “\texttt{dec}” operations can be used to increment (increase by 1) and decrement (decrease by 1) a variable. For example, if the variable \texttt{N} holds 12, after the operation “\texttt{inc N}”, \texttt{N} will hold a 13. If the variable \texttt{N} holds a 12, after “\texttt{dec N}”, \texttt{N} will hold a 11.

The \texttt{int} operator can be used to extract the integer part of a real number. For example,  

\[
\text{int}(2.3) = 2 \\
\text{int}(1.0) = 1 \\
\text{int}(3.99) = 3
\]

Relations and Logical Operations

When writing algorithms, we’ll often need to make comparisons and the standard relations can be used along with the symbols, for example,  

\[
< \quad \text{(less than)} \\
\leq \quad \text{(less than or equal)} \\
> \quad \text{(greater than)} \\
\geq \quad \text{(greater than or equal)} \\
== \quad \text{(equal)} \\
!= \quad \text{(not equal, } <> \text{ can also be used)}
\]

Note that “==” is used for comparison, while “=” is used to assign values to variables.

An expression that can be true or false, such as “\texttt{A < B}” is called a \texttt{logical expression}. Below are some simple logical expressions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; 0</td>
<td>False</td>
</tr>
<tr>
<td>0 &lt;= 0</td>
<td>True</td>
</tr>
<tr>
<td>0 == 0</td>
<td>True</td>
</tr>
<tr>
<td>0 != 1</td>
<td>True</td>
</tr>
</tbody>
</table>

More complex logical expressions can be formed by using the logical operations \texttt{AND}, \texttt{OR} and \texttt{NOT}. Here are some examples.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &lt; 0 AND 1 &lt;= 2</td>
<td>False (both must be true for AND to be true)</td>
</tr>
<tr>
<td>0 &lt;= 0 AND 1 &lt;= 2</td>
<td>True</td>
</tr>
<tr>
<td>0 == 0 OR 1 &gt; 2</td>
<td>True (if one or both are true, OR is true)</td>
</tr>
<tr>
<td>0 == 1 OR 1 &gt; 2</td>
<td>False</td>
</tr>
<tr>
<td>NOT(0 &lt; 1)</td>
<td>False</td>
</tr>
</tbody>
</table>
More about Pseudocode Statements

It is helpful to think of pseudocode statements as instructions that are obeyed by an imaginary computer. These statements are used to perform various actions throughout the life of an algorithm. The language described below has only six different statements. You can add more if you wish, but these six are enough to write most algorithms!

#1

**Assignment statements** are used to give values to variables. Recall that on the first page, we used “N = 0” to indicate that the variable N was to receive the value 0. Various other notational conventions could also be used, but this is the most common and reads easy.

**Examples of simple assignment statements:**

```
Count = 93       // stores a 93 in the variable Count
X = 1.23        // stores a 1.23 in the variable X
Ch = 'A'         // stores a character 'A' in the variable Ch
X = X + 1        // compute X + 1 and save result in X, like increment X
```

Note that you can only have a variable on the left hand side of the assignment operator. The following is considered illegal, \( X + 1 = X \).

**Important**: Here’s the rule used by the computer to perform an assignment statement:

1) First calculate the value of the right hand side.
2) Next store the value in the variable on the left hand side.

#2

**Output statements** are used to display information produced by an algorithm. We’ll assume that an imaginary display device exists. For simplicity, we’ll also assume that the display device outputs information one character at a time. Various notations can also be used to indicate that the value of a variable N is to be displayed. These include:

```
write N
print N
output N
display N
```

For the remainder of this handout, the “write” notation will be used. We will assume that the “cursor” (the screen location where the next character will be output) on our imaginary display device moves one position to the right after an item is written. To move the cursor to the beginning of the next line after an item is written, we’ll use the writeln statement.

**Examples of output statements:**

```
write "Hello World"       //displays “Hello World” on screen, cursor remains on line
writeln N                  //displays the value of N, cursor moves to beginning of next line
write "N = ", N           //displays “N = “, then the value of N, cursor stays on line
```

Each of the lines above has a **comment**, which begins with “//” and stops at the end of the line. Comments are included to help human readers and are ignored by the imaginary machine that executes the algorithm.

#3

**Input Statements** are used to get information from some external source, such as the users (imaginary) keyboard. When a read statement is encountered in an algorithm, the algorithm pauses and allows the user to enter info, after which the info is stored in a variable. Commonly used notation for inputting include “read N”, “get N”, or “input N”. This handout will use the “read N” notation.
#4

**Conditional Statements** allow an algorithm to make decisions, i.e. they must be able to change their course of action based on conditions that exist when the program runs. The if-then conditional statement allows one or more statements to be executed conditionally, that is, only if a certain condition is met. The if-then statement is of the form

```plaintext
if (condition) then
    statements to be executed if condition is true
end if
```

The “end if” serves as a delimiter that marks the end of the group of statements to be executed if the condition is true. The use of the “then” is optional.

Examples of if-then statements:

```plaintext
if (N == 0) then
    write “The value of N is 0”
end if

if (N < 0) then
    write “You entered a negative number, please reenter”
    read N
end if
```

The if-then-else statement is a more powerful version of the if-then statement. It is of the form:

```plaintext
if (condition) then
    statements to be executed if condition is true
else
    statements to be executed if condition is false
end if
```

Example using if-then-else statement:

```plaintext
if (N < 0) then
    write “The value of N is less than 0”
else
    write “The value of N is greater than or equal to 0”
end if
```

#5

**Loops** allow statements to be repeated in order to accomplish a task. The *while* loop allows a group of statements to be repeatedly executed while a condition remains true. A *while* loop is of the form

```plaintext
while (condition) do
    group of statements to be executed while the condition remains true
end while
```

Example: An algorithm that allows user to input a series of positive numbers. When a non-positive number is entered, input stops and the sum of the numbers is displayed.

```plaintext
write “Enter series of positive numbers, enter negative number to end”
Sum = 0
write “Enter the first number => “
read N
while (N > 0) do
    write “Adding “, N “ to “, Sum
    Sum = Sum + N //add Sum and N and store result in Sum
    write “Enter next number => “
    read N
end while
write “The sum of the numbers you entered is “, Sum
```
There are times when the programmer knows in advance how many times a loop body should be executed. The for statement (or for loop) is perfect for these situations, it allows a group of statements to be repeated a definite number of times. It is of the form

\[
\text{for } \text{(ControlVariable} = \text{Initial Value to Final Value)} \\
\text{statements to be executed} \\
\text{end for}
\]

Example: A loop to output the first 10 integers.

\[
\text{for } (N = 1 \text{ to } 10) \\
\text{write } N \\
\text{end for}  \quad // \text{N incremented here automatically}
\]

Note that the for loop is preferable when the number of repetitions is know in advance. Use the while loop if you are constructing a loop that will execute an indefinite number of times.

Another Algorithm Example

An algorithm that displays the sum of digits of an integer entered by the user. For example, if the user enters 123, the algorithm should display:

\[
\text{The sum of the digits of 123 is 6}
\]

\[
\text{write } "\text{Enter an integer} => " \\
\text{read } N \\
\text{write } "\text{The sum of the digits of } \, N \, \text{is} \, " \\
\text{SumOfDigits} = 0 \\
\text{while } (N \neq 0) \text{ do} \\
\text{\quad SumOfDigits} = \text{SumOfDigits} + N \text{ mod } 10 \\
\text{\quad N} = N \text{ div } 10 \\
\text{end while} \\
\text{write } \text{SumOfDigits}
\]