Writing Program Algorithms Using Pseudocode

Pseudocode represents the statements of an algorithm in **structured English**. It is English, which has been formalised and abbreviated to look very much like high level computer languages.

There is currently no standard pseudocode. Textbook authors seem to adopt their own special techniques and set of rules, which often resemble a particular programming language.

The aim of pseudocode is so all programmers, regardless of the programming language they choose, can use it. Typically pseudocode has certain conventions, as follows:

1. Statements are written in simple English.
2. Each instruction is written on a separate line.
3. Keywords and indentation are used to signify particular **control structures**.
4. Each set of instructions is written from top to bottom **with only one entry and one exit**.
5. Groups of statements may be formed into modules, and that group given a name.

How to Write Pseudocode

When designing a solution algorithm, a programmer needs to keep in mind that a computer will eventually perform the set of instructions. That is, if the programmer uses words and phrases in his or her pseudocode that is in line with basic computer operations, then the translation from the pseudocode algorithm to a specific programming language becomes quite simple.

Earlier in the file “Writing Programs”, several steps in the development of a computer program were presented. The very first step, and one of the most important, is the defining of the problem. This involves the careful reading and re-reading of the problem until the programmer understands completely what is required. As an aid in this initial analysis, the problem should be divided into three separate components:

Input, or source data provided to the problem;
Output, or end result, which is required to be produced;
Processing or a list of what actions is required to be performed.

The following common words and keywords may be used to represent, **in pseudocode**, the basic computer operations of;

- Getting input,
- Processing the data and
- Outputting the result.

Each operation can be represented as a straightforward English instruction, with keywords and indentation to signify a particular control structure.

1. Receiving Input or Data

When a computer is required to receive information or input from a particular source, whether it be a terminal, a disk or any other device, the verbs **Read** and **Get** are used in pseudocode.

For example typical pseudocode instructions to receive information are:

Read student name or
GET number1, number2

Each example uses a single verb, *Read* or *Get* followed by one or more nouns to indicate what data is required to be obtained.

2. Giving Out or Outputting Data

When a computer is required to supply information or output to a device the verbs **Print**, **Write** or **Output** is used in pseudocode. Typical pseudocode examples are:

Print student number

Write program completed
OUTPUT name, address and postcode

In each example, the data to be written out is described concisely using mostly lower case letters.

3. Processing the Data

a) Performing Arithmetic

Most programs require the computer to perform some sort of mathematical calculation, or formula, and for these, a programmer may either use actual mathematical symbols, or the words for those symbols. For instance, the same pseudocode instruction can be expressed as:

Add Number to Total
Divide TotalMarks by StudentCount

Or

To keep in line with high level programming languages, the following symbols can be written in pseudocode:

+ for Add
- for Subtract
***** for Multiply
/ for Divide
= for equal
( ) for Brackets

Some pseudocode examples to perform a calculation are:

Total + Number
TotalMarks / StudentCount
(F-32)*5/9

b) Assigning A Value To A Variable

There are many cases where a programmer may need to assign or store or save a value or result in a variable.

To give a variable an initial value in pseudocode the verbs **Initialise** or **Set** may be used.

To **assign** a value as a result of some processing the symbol '=' or '←' is written.

Some typical pseudocode examples are:

Initialise total accumulators to zero or
Set total accumulators to 0
total accumulators ← 0 or
total accumulators = 0
TotalPrice = Costprice + Sales tax or
TotalPrice ← Costprice + Sales tax or

c) Processing Data Using the Three Control Structures.

This will be covered in detail in the next files "**Control Structures and Pseudocode .doc**".

APPENDIX A

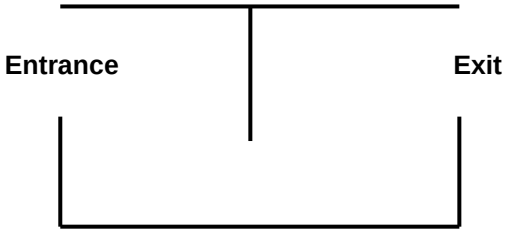
A solution algorithm for a robot to navigate through any maze

(Note this Maze algorithm is written in pseudocode)

An Amazing Algorithm

We wish to write a computer program that can be used by a robot to guide itself through **any** maze. Let us assume that the robot's hardware allows it to respond to the three simple instructions:

1. **Step forward**
2. **Turn left** and
3. **Turn right**



To instruct our robot to navigate the simple maze at the right, the following sequence of instructions can be given to the robot in order for it to guide itself from entrance to the exit:

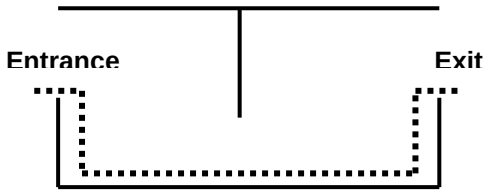
Step forward turn right
Step forward turn left
Step forward turn left
Step forward turn right
Step forward

The problem with the above algorithm is that the solution can only be used for that maze. We will need a different series of instructions for each possible maze that the robot may encounter. This algorithm is unacceptable because we want the robot to guide itself through **any** maze, so the problem must be approached from a different perspective. A generalised technique that will allow the robot to enter any maze and, by using simple logic, find its exit must be found. In programming terms we are looking for an algorithm that will guide a robot through mazes.

Such an algorithm for navigating through mazes is already well known and is commonly known as a "wall hugging algorithm". Stated in English it goes like this:

Have the robot enter the maze and move along with its right side touching the wall of the maze until it reaches the exit.

What this amounts to is that the robot should walk through the maze hugging the wall to its right. For the previous maze this produces the following (successful) motion:



Unfortunately, however, our robot doesn't understand the complete English language. Therefore we need to translate the algorithm from **General English** into **Structured English** using the simple instructions that the robot does understand which are **step forward**, **turn left** and **turn right**. At this point we also need to assume that the robot is capable of answering the yes/no questions: "Am I facing a wall?" and "Have I reached the exit?"

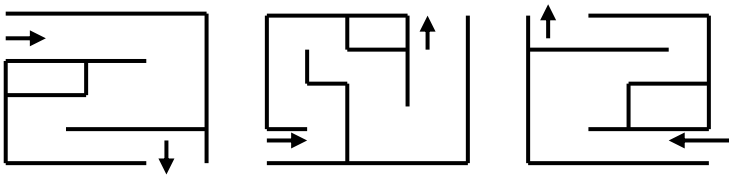
If we assume that the robot always starts by facing into the entrance of the maze then the algorithm will begin with the following instruction in order to move the robot into the maze:

Step forward

Once inside, the next thing the robot must do is **rotate** in such a way that it **puts a wall on its right hand side**. Our robot doesn't have an instruction called "**rotate so that there is a wall on your right hand side**" but it is can make a **decision** based on whether it is facing a wall or not since it has sensors to do this. Therefore we can build this instruction out of the simpler instructions, like so:

```
Turn right

While facing a wall?          {Note: "facing a wall" is a decision}
    Turn left
```

What this says is that in order to "rotate so that there is a wall on your right hand side" all the robot need to do, for any maze, is turn right once, and then as long as it is still facing a wall keep executing the command turn left. Consider the sample mazes that follow, and prove to you that this technique works.



Our algorithm so far, in terms that the robot can understand, is as follows:

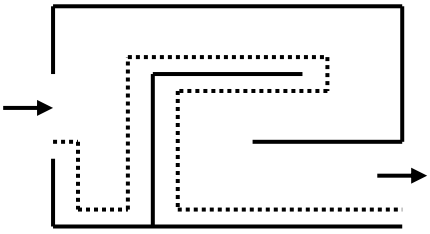
```
Step forward          {move into the maze}
Turn right            {place a wall to our right}

While facing a wall?  {keep turning left until you are no longer facing a wall}
    Turn left

Step forward          {move into the maze}
```

Now all the robot has to do is keep on repeating all but the first line of this algorithm until it escapes the maze! So our completed algorithm for traversing any maze is as follows:

```
Step forward          {move into the maze}

Repeat
    Turn right         {place a wall to our right}

    While facing a wall?   {keep wall to our right}
        Turn left

    Step forward           {move into another room)
Until outside the maze
```

Notice how simple this algorithm is. It reads almost like English, and yet it allows our robot with its very simple abilities to traverse any maze **that has a solution**.

The diagram on the right shows the path taken by the robot when traversing another maze. Ensure that you understand why it chooses the path shown.



As an exercise, refer back to the sample mazes presented earlier and deduce what path the robot will take through those mazes.

This right wall-hugging algorithm that we have developed is not the only algorithm that can be used to successfully traverse mazes, of course. A left wall-hugging algorithm would have worked just as well. Can you think of any other algorithms?

Do Activity 4 – Writing Program Algorithms Using Pseudocode

END