# Turing Machine Development Environment

Daniel Guetta – guetta@cantab.net

## 1 – Abstract

Turing Machines are the basis of the computability theory and of computer science. They are chiefly used to decide what a "normal" computer can and cannot do. However, they are very hard to program and few aids exist. This project is a new and simple development environment for Turing Machines.

## 2 – Introduction

Invented by the British mathematician Alan Turing in the early 1900s, Turing Machines were originally only built to define the word "algorithm" – a great challenge of the 20th century, laid out by the mathematician David Hilbert in his address to the International Congress of Mathematicians in 1900[i]. However, since then, Turing Machines have evolved to become one of the simplest models of the modern computer – that is to say, whatever modern computers can do, Turing Machines can do and whatever Turing Machines can do, modern computers can do[1]. They are equivalent. This theory of equivalence between Turing Machines and modern computers is now part of the Church-Turing thesis[ii].

Since the entire program is based on the theory of Turing Machine, we will describe this theory at length in the introduction and refer to it later in the report.

Turing Machines consist of a tape limited on one side but infinite on the other (semi-finite tape) and a read/write head (see figure 1).

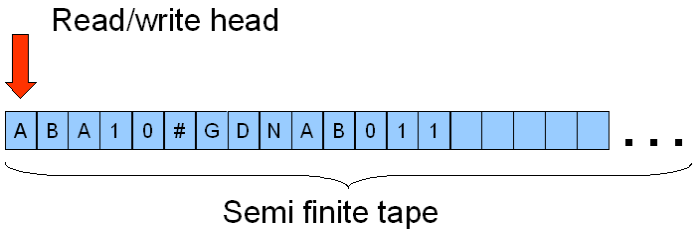



*Figure 1 – The Turing Machine*

The tape is divided into cells and each cell can contain one symbol only. However, a symbol could be more than one character (for example `bigSymbol` could be a symbol). A special symbol, the blank symbol, is used to fill every empty cell

The read head can move left and right anywhere over the tape, one cell at the time. When it is immediately over a cell, it can read the symbol there, write a new symbol (overwriting the one already there) or simply do nothing[iii].

Telling Turing Machines what to do (*programming* Turing Machines) involves a set of named states, which are specified in the building stage of the machine. During the running of the machine, the machine is always in one state only. We program the machine by giving it a set of instructions in the following format:

```
If machine is in state x and on a cell with symbol y, move to state a, write symbol b
                in the current cell and move left, right or not at all.
```
[iv]

The set of instructions is known as the *delta value (δ-value)* of the machine. Please note that the new state could be the same as the current state and the new symbol could be the same as the current symbol, if no change is desired.

[1] An example of something computers cannot do (which quite surprising) is decide whether another program will ever halt. It is impossible to write a program that will examine another program and say whether it will halt or will continue looping for ever (crash) – Turing Machines cannot do it, and therefore modern computers cannot do it either.

It can easily be seen, however, from the format above, that the programming of a Turing Machine can be extremely tedious, especially when the number of states and symbols rises above a certain level. Furthermore, finding problems in Turing Machine programs and fixing them (*debugging* the programs) – which can involve tracing through the actions of the machine – can be very monotonous and time-consuming.

It is for those reasons that some different ways to represent the δ-value than lines of instructions were devised. One of them is the use of a truth table, where each column represents a symbol, each row represents a state and each cell represents one instruction. Another method is to have diagrams where each box on the diagram (*node*) represents a state and each connecting line an instruction[v]. Figure 2 shows the same simple Turing Machine (that writes alternate 1s and 2s on the tape and never stops) represented in three different ways, to allow you to contrast between the techniques. 0 is the blank symbol in that machine.

**Method 1 – Instructions**

If in state **Q1** over symbol **0** go into state **Q2**, write symbol **1** and move **right**.
If in state **Q2** over symbol **0** go into state **Q1**, write symbol **2** and move **right**.

**Method 2 – Truth Table**

| | **0** | **1** | **2** |
|---|---|---|---|
| **Q1** | o into **Q2**, write **1**, move **RIGHT** | - | - |
| **Q2** | Go into **Q1**, write **2**, move **RIGHT** | - | - |

**Method 3 – State Diagram**

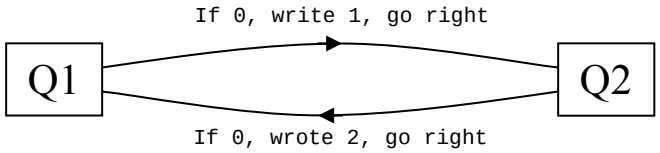



*Figure 2 – Comparison of representations of Turing Machines*

From now on, instructions in the format above will be represented as `Current state, current symbol, new state, new symbol, new direction` for the sake of brevity.

Another part of the theory of Turing Machines worthy of attention is the topic of variations. So far, the only type of Turing Machine (*model*) explored is that of a single tape and a single read head. However, there are several variations to this model. One of the most interesting ones introduces more than one tape and gives each tape and individual read head, which can move independently from others (see figure 3). That it to say, at each step, both heads can write different symbols on the different tapes and then move different directions. However, both read heads are always in the same state. To program this model, the instructions are slightly modified. For example, for a three-tape machine:

If machine is in state **x** and on cells with symbols **y_1, y_2 and y_3**, move to state **a**, write symbols **b_1, b_2 and b_3** in the current cells and, for tape 1 move **left**, **right** or **not at all**, for tape 2 move left **left**, **right** or **not at all** and for tape three move **left**, **right** or **not at all**.

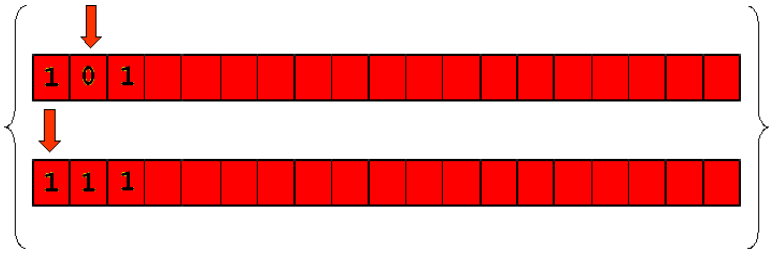



*Figure 3 – A two-Tape Turing Machine*

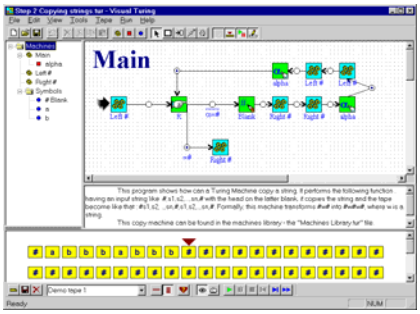
*Please note the braces are used only to show the machines share the same state – they have no other significance.*

It may look like the multiple-tape Turing Machine model can carry out (*execute*) some programs that the ordinary single tape model could not – in other words, it may look like this model is more powerful than the normal one. However, this is not so – they are entirely equivalent. This can be easily proved by showing how any step done by the two tape machine can be done by the single tape machine (*simulation*). Since this simulation is extensively used in the program, it is covered in detail in the design rather than here.

## 3 – Previous work

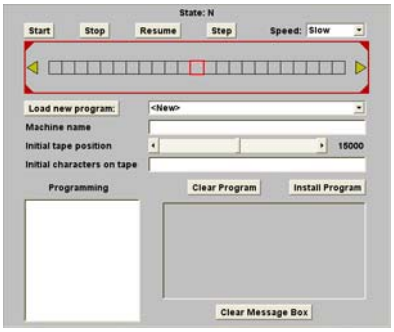
Before outlining the aims of our program, we will now examine previous work in the field to evaluate the need of various features in the program.

1. **Cheran Soft Visual Turing**[vi]




Visual Turing is a freeware program that allows the drawing of Turing Machines as state diagrams and to run those state diagrams in a simulator. The program also allows the user to pause in a given place. However, the program has three main disadvantages. First of all, there is no way to convert those state diagrams into truth tables or instructions. Second, the tape given is finite and allows too few cells – as soon as a slightly complicated Machine is run, the tape runs out. Finally, the model proposed is not a "proper" Turing machine – it allows function calling and variable declaration, which traditional machines do not support without added programming.

2. **Various applets on the web**[vii]




There are several applets available on the web to program Turing Machines, but all have the same features; the input of instructions in the abbreviated form described above and the simulation of those instructions. In that respect, they are useful to carry out a simulation that a student would otherwise have to carry out by hand, but they offer little more capabilities than a human would.

There is more software available on the market (for example, Stanford's excellent "Turing's World" suite[viii]), but all provide functionality similar to the two programs above.

## 4 – Aims of the Program

The aims of the program attempt to keep the solid foundations laid down by previous work while building up on them and improving them.

1. The ability to program Turing Machines as truth tables, state diagrams and simple instructions and to convert between all three formats.
2. The ability to convert instructions from pseudo code[1] to Turing Machine instructions
3. The ability to convert multiple tape Turing Machines to single tape Turing Machines – in other words, to simulate multiple-tape Turing Machines on a single-tape machine.
4. The ability to simulate Turing Machines in the formats above. The Simulator must have a Graphical User Interface and must give real-time visual feedback to the user, with a possibility to pause on a state for debugging.
5. The ability to save Turing Machines and re-use them later.
6. The ability to view, simulate and edit several machines at the same time, for comparison.
7. The program must be constructed so that other programs can use its parts easily. For example, if another program wants to simulate a Turing Machine, it should be able to simply take the "simulator" control from the project and use it, without needing the other parts of the project.

## 5 – Development environment and Program type

The development environment used was be Microsoft Visual Basic 6, chosen for its capability to produce professional Graphical User Interfaces – a key part to the project – and for it's interpreter, which allows to modify code while the program is being run, thereby greatly simplifying debugging. The only downside to Visual Basic is the slow speed of its program *(its low performance)* compared to languages like C or C++. However, the aims of this project mention nothing about performance, and the acceptable performance of VB is sufficient for this program.

The program has a graphical user interface, as mentioned in aim 4 above.

The program also has a multiple-document-interface – which means it will have the ability to open, edit and simulate several machines at the same time but all within the same program.

## 6 – Program Structure

The structure of the project is purely modular (as mentioned in aim 7 above) – this means that the project split up into several well-contained parts (*controls*), and each of those controls could work by themselves, even without the other parts. However, they all have the common point that they need to store a Turing Machine.

The method of storage of Turing Machines in the different controls was changed halfway through the project. At the start, each part of the project had its own version of the machine stored within itself, as shown in figure 4. All the controls, however, held exactly the same machine but just displayed it in a different way, so all the stored machines were exactly equal.

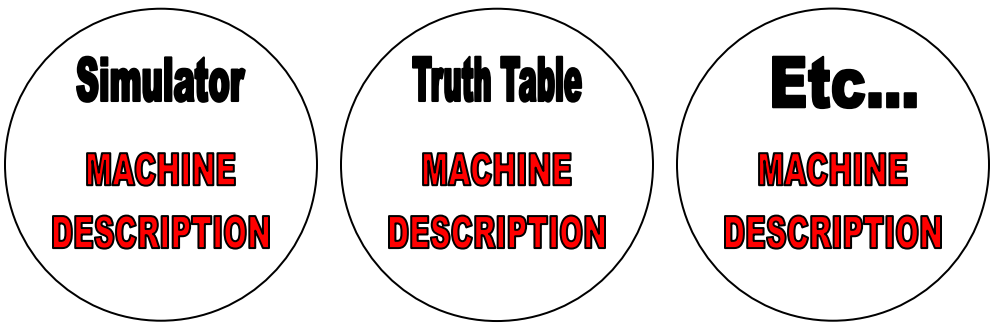



*Figure 4 – Old program structure.*
*Each part of the project holds its own version of the machine.*

[1] Pseudo code is a language between normal English and programming language

However, this method proved extremely inefficient for two reasons:

1. **An enormous amount of memory was wasted**. Since there was a copy of a Turing Machine in each control, multiple copies of the machine were kept, even though the machines were exactly the same – this used up a large amount of memory.
2. **Inconsistencies could develop between machines**. If one part (say, the Truth Table) updated their machine (as a result of a change by a user, for example), there was a chance that the machines in other controls were not updated, thereby causing different parts of the program to hold different machines. This could result in serious problems. For example, the Truth Table might be displayed the Truth Table of a machine different to that being simulated in the simulator.

The structure was therefore changed to that of figure 5. The machine is now stored in one central place (an instance of the `clsMachine` class[1]) and each control holds a pointer to that central storage position. Whenever one of the controls wants to update a machine, or display it, it simply uses this central machine. This gets rid of both problems above. Another advantage to this method is that a program can have a machine separate from any control – a control is not needed to store a machine.

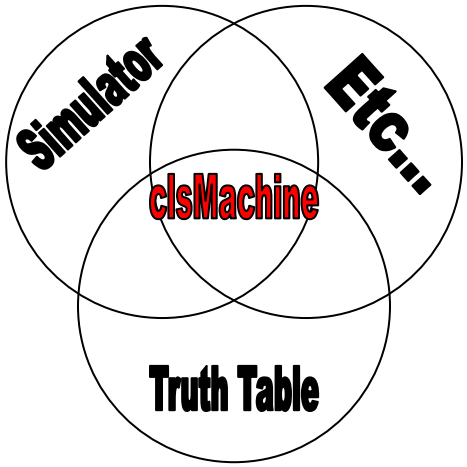



*Figure 5 – Current program structure.*
*A centralised storage location of all controls*

We will now cover `clsMachine` class, of which the central storage location is an instance, in more detail.

### 6.1 – The `clsMachine` class

An instance of the `clsMachine` class[2] is used as a central storage location for the machine used by the program. It makes available to the user (*exposes*) methods and properties which allow the user to create and modify Turing Machines. For example, it exposes an `addState` method, which adds a state to the machine. All modifications to the machines are made through those properties and methods.

Once this class is covered in detail, there is no need to review each component one by one (except for the conversion component discussed below). This is because they are all simply displaying the machine differently, but chiefly just using the `clsMachine` class.

The way data is stored within the `clsMachine` class is different for each component.

---

[1] The meaning of *class* and *instance* is dealt with in the next footnote.

[2] To understand the meaning of *class* and *instance*, we can take the example of a commercial bicycle firm. They first draw plans for a new bicycle (including things about it – its *properties*, and things it will need to do – its *methods*). This plan is equivalent to the class declaration. Then, from that one plan, the firm will make many bicycles. Each bicycle is equivalent to one instance of the class. Each bicycle, however, could have a different colour since colour is a property and properties vary with instances. However, properties apart, the basic structure will stay the same for all bicycles. This is the same with our Turing Machines – each instance of the `clsMachine` class will have different sets of instructions, for example, but the basic definition of a Turing Machine will stay the same in each.

### 6.1.1 – For states

All states are held in one array[1] - `Q()`. The array is of the user defined data type `QType`[2]. When a state is added, it is stored at the next available storage space in the array.

The name of the state (its `ID`) is stored within the user defined data type, and the state is always referred to by its `ID` outside the class or in class methods and properties. However, within the class, the state is referred by its array position only. So for example, if a state called "Q1" was stored in array position 3, any control communicating with the class would use "Q1" and the class would answer "Q1". However, within the class, 3 would be used (see figure 6).

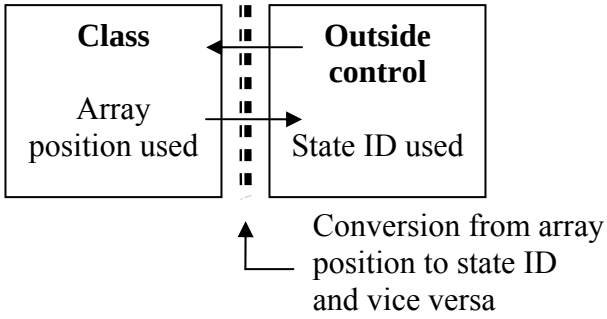



*Figure 6 – State referencing in the class*

This has two major advantages:

1. It is much quicker to access a state by its array position than by its state ID (since doing it by its state ID involves some kind of search). Therefore, the class runs much faster.
2. State IDs can be changed at will – within the class, only the array position is used and a change in the name will have no effect, since the array position would stay the same. If the state ID was also used within the class, changing it would involves going through every single δ-value and changing the ID there.

However, the state ID still needs to be converted to an array position at least once, when it goes from one to the other. There are two methods available for this, each with their relevant advantages:

1. Search through the entire state array every time you need to find an array position[3]. Whenever you find an ID that matches the one you are looking for, you know you found the right position. This method is advantageous because it doesn't use much memory, but it is problematic because searches are very slow.
2. Build a table which allows you to go straight to the value itself (a *hash table*). This is advantageous because finding the value would then be extremely fast, but it is problematic because hash tables take a large amount of space in memory.

This dilemma is a typical time-memory trade-off – either the time or the memory. For this program, we have chosen to use the hash table option, since it greatly increases the speed of the program. The extra amount of memory taken justifies the sacrifice of time.

A final point that needs to be covered is the deletion of states. When a state in the array is deleted, there are two possible options. Either every state below that state is shifted up to fill the gap (remember, arrays are tables), or a garbage value is written to the state position to show it is deleted (see figure 7).

---

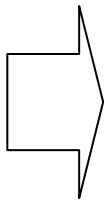
[1] Arrays are tables of data

[2] A user defined data type (UDT) allows the programmer to take many different variables of many other types (integer, string, etc…) and to combine them into one big type. For example, the `QType` mentioned above contains three strings, all put together in one type.

[3] Or use an optimized searching algorithm – but this would still take a long time

**Method 1 – Shifting array contents**

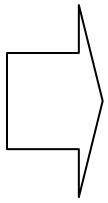
| Array pos. | Contents |
|---|---|
| 1 | State 1 |
| 2 | State 2 |
| 3 | State 3 |
| 4 | State 4 |



| Array pos. | Contents |
|---|---|
| 1 | State 1 |
| 2 | State 3 |
| 3 | State 4 |

**Method 2 – Writing a garbage value**

| Array pos. | Contents |
|---|---|
| 1 | State 1 |
| 2 | State 2 |
| 3 | State 3 |
| 4 | State 4 |



| Array pos. | Contents |
|---|---|
| 1 | State 1 |
| 2 | ######## |
| 3 | State 3 |
| 4 | State 4 |

*Figure 7 – Two methods of deleting the array element highlighted in grey*

Even though the first method looks much more appealing, since it leaves a "clean" array, it has two major flaws which make it very hard to use for this program:

1. The shifting of array elements back up the table can be very time-consuming since every element has to be moved one by one. Even if special API functions that move the entire block as a chunk are used, the process is still slow.
2. If we shift array elements, the array pointer for every element changes. This implies that we need to update the array pointer in every part of the class. However, it is to avoid this that we chose array pointers in the first place!

Therefore, the second method is used. However, it is slightly improved so that not too much space is wasted. When a user adds a new state, the program will not immediately add it to the end of the array. It will scan through every cell until it find one which has a garbage value, and this is where it will put the new state, thereby recovering the lost space. The only time the program will enter the value at the end of the array is if there are no garbage values at all. Even though this search for a garbage value will also take time, it is much quicker than the other method, which would imply searching through the entire δ-array.

The class holds the array positions of the starting states and final states in two simple variables. When they are set or accessed, the array position ⇔ State ID conversion operates.

### 6.1.2 – For symbols

Symbols are stored exactly like states in every respect except for one. When a symbol character needs to be converted to a symbol array position, no hash table is used but the entire array is searched. The reason why this is acceptable for symbols but not for states is that there will usually only be a handful of symbols in the machine, while there may be many states. Hence, going through the entire symbol array would take a very short time, while doing the same for the state array would take much longer.

The class holds the array position of the blank character in a simple variable. When it is set or accessed, the array position ⇔ symbol conversion occurs.

### 6.1.3 – For instructions

The storage of instructions is slightly more complicated than it seems. It would seem logical to simply store the instructions as a Truth Table in a 2 dimensional array[1]. However, the problem with this method can easily be seen by looking at the example truth table above (second page of this report) – most of the

[1] The simple list of items seen so far is a 1 dimensional array. A two dimensional array is a "normal" table.

cells are blank. This means that an enormous amount of memory would be wasted by useless white cells (see figure 8). This is especially true when we deal with multiple-tape machines, where the number of columns in a truth table is equal to *Number of characters*$^{Number\ of\ tapes}$ and only a very small number of those columns are used.

| | A | B | …. | *n* |
|---|---|---|---|---|
| $Q_1$ | - | - | - | - |
| $Q_2$ | trans | - | - | trans |
| … | - | - | - | - |
| $Q_n$ | - | - | - | trans |

*Figure 8 – Storing a truth table in an array*
*Even though only three cells are used, many more are stored and take up memory*

There are two other methods which we considered to store the instructions:

The first method involves using a structure called a *very sparse array*. It is designed for arrays like ours where there can be thousands of cells in total, but only a few with values written in them. It works by keeping the column headers in a linked list[1] and the row headers in a linked list. From each header, there then comes out another linked list containing all the items in that row/column. Figure 9 illustrates the concept visually[ix].

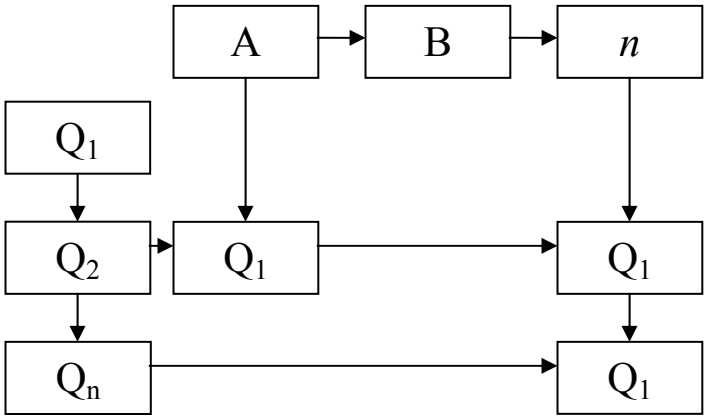



*Figure 9 – A very sparse array*

The problem with a very sparse array, however, is that every time a cell needs to be accessed, the program must search through the entire linked list for the row/column header, and then the entire header linked list for the cell. With an array, direct access would have been possible.

The second method, which is the one used in the project, uses a 1-dimensional array. There is one cell in the array for each state. The cells themselves then contain a list. To find a given instruction in the list (given the current state and current scanned character), the steps would be the following:

- Directly access the correct list in the array for the state we want. For example, if we need state $Q_1$, go to the array position for state $Q_1$.
- Once the list for the correct state is found all we need to do is to quickly search trough the very short list to find the instruction for the character we want.

[1] A linked list is a special kind of list where each item holds a pointer to the next item, but each item is stored individually. This means that the only way to get to an item is to go to the item before it and follow the pointer. This also means that whenever an item needs to be accessed, we need to start from the first item in the list and go down the list until we find the item we want – there is no way to directly access the items. There are advantages to a linked list, but they are out of the scope of this article. Citation ix provides excellent reference on the topic.

| Array position (= state array position) | Contents |
|---|---|
| 1 | • A → $Q_1$, 2, L<br>• B → $Q_1$, 3, R<br>• C → $Q_2$, 4, S |
| 2 | • 2 → $Q_1$, 2, L |
| 3 | • C → $Q_1$, 2, L<br>• B → $Q_1$, 3, R |

*Figure 10 – The solution used in the program*
*No space is wasted, since every state will have at least one transition*

**7 – Conversion from a multiple-tape machine to a single tape machine**

In the introduction, we mentioned that multiple-tape machines were as powerful as single-tape machines, and that it could be shown by simulation. There are many simulations we could use, but we chose one found in a book by Michael Sipser for its practicality of implementation. This is the simulation we use in the program to convert multiple-tape machines to single tape machines. We will now explain the simulation in detail.

First, the several tapes of the multiple-tape machine, complete with read heads, need to be represented on a single tape machine. The best way to do this is to write each tape one after the other on the single tape machine, but to separate them by **$** symbols, for example, to show the separation between tapes.

To show where the read head is on each tape, a more subtle trick has to be used, in which every symbol in the machine is doubled, but with a ‘ character appended to it (remember symbols can be more than one character long). Thus, a machine with symbols `A, B, C` would now have symbols `A, A′, B, B′, C, C′`. Then, to show the read head is over a certain symbol, we add the ‘ to simulate the read head. Therefore, if we had the tape:

`A B C C A`

with the read head on the second C, the tape would now be:

`A B C C′ A`

A final step needed to simulate the multiple-tape machine is to add > an < symbols at the beginning and end of the entire simulation respectively. If this was not done, the machine would not know whether the symbol after the last “visible” $ was the end of all tapes, or the beginning of a new tape. Since tapes are infinite on the right, this blank symbol could well be the beginning of another tape with a huge number of blanks. Figure 11 should clarify the entire process by giving an appropriate representation of the machine in Figure 3.

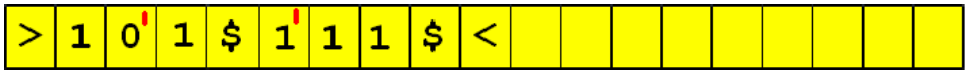



*Figure 11 – A simulated multiple-tape machine*

Once we have this representation, simulating the actions of the multiple-tape machine on the single tape machine is relatively simple to describe, but extremely hard to program:

1. Sweep from left to right over the tape, remembering each symbol under the read head by entering a new state for each different symbol under the read head (figure 12). Once all the symbols have been scanned, it is clear what transition is needed.
2. Sweep back to the left, changing the symbols and the position of read heads each time one is met. If the machine ever needs to go right at the end of a tape, simply shift the entire tape content one cell to the right to make space

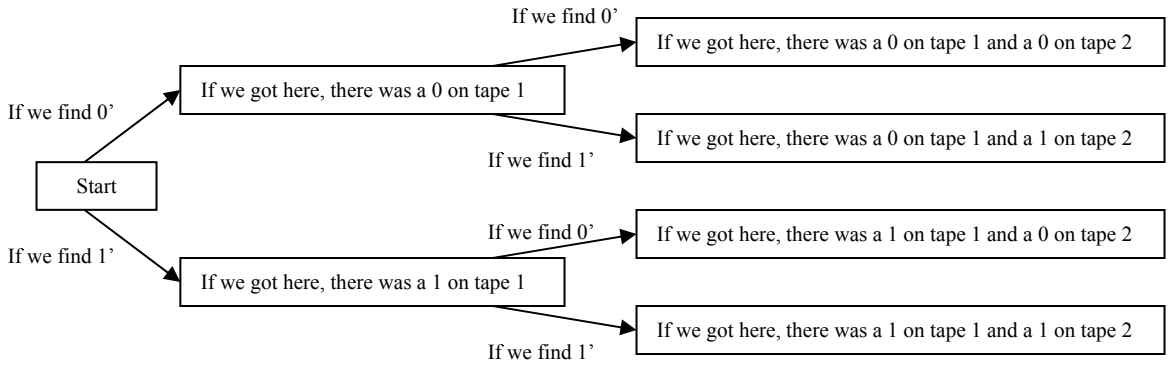



*Figure 12 – "Remembering" the characters we scanned*
*Each box is a state*

**8 – Conversion from Pseudo code to a Turing Machine**

One of the most interesting and complicated aims of the project was aim 2 – the pseudo code conversion. This conversion implied having a compiler that could take any normal high level language code and compile it to simple Turing Machine instructions. However, there was no aim specifying a conversion from simple Turing Machine instructions to pseudo code. The reason for this is simple. Inasmuch as conversion from higher level instruction to lower level ones is feasible and has already been done by all compilers existent today (only for normal computers, though – it has never been done for Turing Machines), the reverse – a process called reverse engineering – is extremely hard and is still an open challenge of computing. If this process was simple, then the entire computing industry would be revolutionised – Windows could be reversed engineered and reproduced by rivals of Microsoft with great ease and programs would no longer be safe from copying by rival companies. Therefore, the conversion from Turing Machines to high level code is not covered.

This section of the project was unfortunately not covered due to lack of time and people! However, the theory for the conversion was laid down and is discussed below, assuming a language with the following features:

- Loops
- Conditional blocks (if statements)
- Explicit declaration and use of global and private variables
- Function calling (and returning values)
- The four basic arithmetic operators and indices (including expressions involving brackets)

This section might, however, include some technical information that we will be unable to explain due to lack of space.

The machine to emulate a computer would contain 4 tapes. One tape for user defined variables, one tape for system variables, one tape for the call stack and one tape for processing. Before compilation to a Turing Machine, the source code would have to be parsed and converted to a tree, which would allow the entire code to be executed by the following seven functions only:

- Addition
- Subtraction
- Division
- Multiplication
- Equation
- Variable retrieving
- Variable assignment

For example, the following code would be converted into the tree in figure 13.

```
If ((a + b) * c) = 5 Then
  c = d + f
End if
```

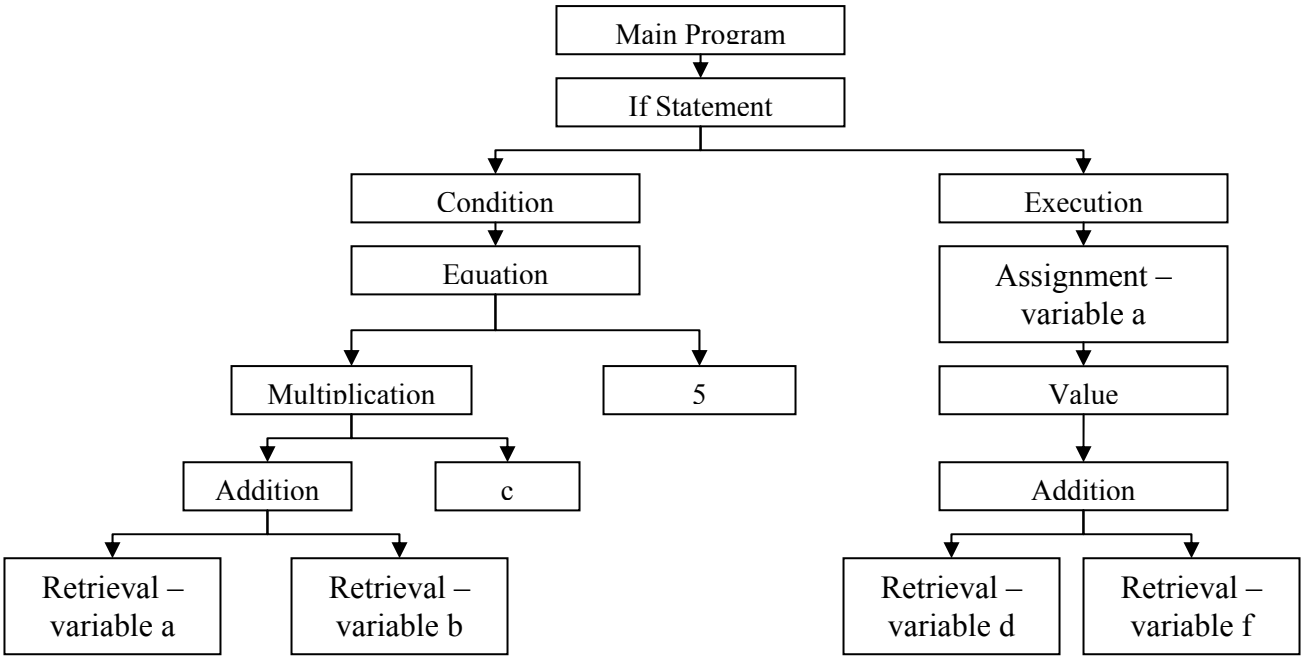



*Figure 13 – A tree for the aforementioned source code*

We will now examine each of the seven items and describe methods to implement them with Turing Machines. Then, we will examine the different high-level language construct and describe how to convert them to the trees described above.

**8.1 – Addition**

Addition is extremely simple. Assuming numbers are given in a unary format, all that needs to be done is the copying of the second number to the first number, using a simple copying machine[x]. In the context of the architecture of our multiple-tape machine, two new system variables could be created on the system variables tape, one with each number (see sections on variables below). The copying machine would then concatenate both variables.

**8.2 – Multiplication**

Multiplication could also be achieved using a simple copying machine. The number, again in unary format, would be copied into a result variable repeatedly until the end of the multiplication. For example, in the multiplication 5 × 3, the unary number 11111 would be repeated three times into the result variable.

**8.3 – Subtraction**

Subtraction is somewhat more complicated to implement. Let us take $a - b$ as an example. The simplest approach is to take both the numbers in unary format and to cross a 1 from $a$ for every 1 from $b$. The resulting number left in $a$ will be the result of the subtraction. If negative subtraction also needs to be supported, the Machine could check, every time it crossed a 1 from $a$ whether $a$ was now equal to 0. If that was the case, it could write a – sign at the start of the tape and add 1s instead of removing them thereafter.

**8.4 – Division**

Division is also relatively complicated to implement. The two numbers would, again, have to be stored in unary format in two system variables. Then, the machine would repeatedly try to remove a copy of the second number from the first, keeping count of the number of removals. For example, for the division 10 ÷ 5 (coded 1111111111 ÷ 11111), the program would repeatedly try to remove 5 1s (11111 – the encoding for the number 5) from the encoding for 10 until it ran out. When it runs out, the number of removals carried out is equal to the result of the division.

### 8.5 – Equation

Equation is a relatively simple but tedious process. Every character in one string must be compared with a character in the next string. Once the two characters are compared:

- If they match, they are deleted, and the entire process starts again from the next characters (which are now the first characters of the string)
- If they do not match, the equation should return false immediately

Once either string runs out, the machine would then check the status of the other string – if it ran out at the same time, the machine would be accepted – otherwise, rejected.

### 8.6 – Variable storage issues

The issue of variables is more complex than other ones. First, we will look at the method used to actually store variables on the variable tapes. Each variable is written on the tape and separated by a unique variable symbol. For example, if three variables *a, b* and *c* with values "hello", "goodbye" and "end" respectively were to be stored on a tape, the tape would look like the one in figure 14.

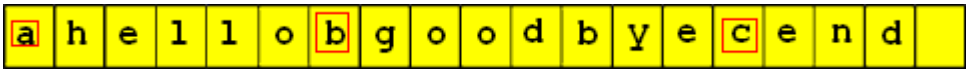



*Figure 14 – Storing variables on a tape*

*Notice how the variable symbols are given special markers so the machine does not confuse them to be part of the actual variables*

The storage of private variable would include an extra step to this storage model. For every scope block, there would have to be a start and end marker, and within those two markers, the variables in that scope would be stored. Garbage collection is not discussed here for lack of space.

#### 8.6.1 – Variable assignment

Assigning a variable would require the following steps:

- On the relevant tape (system or user defined) find the variable marker – if it does not exist yet, create it at the end. When scanning the tape, skip any blocks which are out of our scope.
- Copy the value to be assigned straight after the marker character
- If there isn't enough space between the marker character and the start of the next variable, shift the tape contents using a simple shifting machine[x].

#### 8.6.2 – Variable retrieval

Retrieving a variable would require the following steps:

- On the relevant tape (system or user defined) find the variable marker – if it does not exist yet, return an error (implicit variable declaration is not supported by our pseudo code language). When scanning the tape, skip any blocks which are out of our scope.
- Copy the value between the marker found to the next marker (the variable) from the variable tape to the desired location (usually the processing tape)

### 8.7 – Loops

Loops are fairly easy to convert to a tree using only the features specified above.

- First, create a system variable to use as a loop counter (unless an existing variable has already been specified in a `For` loop declaration). In both cases, initialise it with the initial value needed.
- Run the code within the loop repeatedly, incrementing the counter each time.
- Each time the code is about to run, compare the target value and the loop counter value using the code specified in section 8.5 above. If they match, exit the loop.

### 8.8 – Conditional blocks

With conditional blocks, each expression of the condition would first need to be sub-split into the seven basic commands described above. That done, the different expressions would be evaluated and equated. The code within the block would then be run depending on the result of the equation.

### 8.9 – Function calling

Function calling would be fairly simple, assuming the binary encoding of the Turing Machine of the function was stored in a normal system variable. The steps to calling would be:

- Write the current tape configuration of the processing tape to the call stack tape.
- Run a universal Turing Machine on the binary encoding of the Turing Machine of the function – for any variables needed by the function, create a new scope block. Make the first variable in the new scope block the return value for the function.
- When the function is done running, read the tape configuration from the call stack tape and restore it. Get the return value from the first variable in the function scope block and then delete the scope block created.

### 8.10 – Arithmetic operations involving brackets and indices

Arithmetic operations involving brackets are simple – all that needs to be done is to split the bracketed operation into the seven basic operations specified above and store the results of each of the brackets in a system variable. All operations on that bracket can then be done on the system variable it returned.

Supporting indices is also simple – all that needs to be done is a series of multiplication on the relevant number or variable.

## 9 – Evaluation

We will now take the aims one by one and see how far, if at all, our programs has fulfilled them.

1. The ability to write Turing Machines as truth tables, state diagrams and simple instructions and to convert between all three formats.

   The programme fulfilled part of this aim. All three formats are implemented in the program and can be converted to simple instructions, but the conversion **to** state diagrams has not been implemented due to lack of time.
2. The ability to convert multiple tape Turing Machines to single tape Turing Machines – in other words, to simulate multiple-tape Turing Machines on a single-tape machine.

   This aim was fully implemented. The conversion procedure described above is used and the results are graphically relayed to the user.
3. The ability to simulate Turing Machines in the formats above. The Simulator must have a Graphical User Interface and must give real-time visual feedback to the user, with a possibility to pause on a state for debugging.

   This aim was fully implemented. The tape is presented as a series of squares and constant graphical feedback is given to the user concerning the current state and tape contents.
4. The ability to save Turing Machines and re-use them later.

   This aim was also fully implemented through the production of sequential **.vtf* files that can be written and re-read by the program.
5. The ability to view, simulate and edit several machines at the same time, for comparison.

   This aim was fully implemented by the multiple-document-interface used.
6. The program must be constructed so that other programs can use its parts easily. For example, if another program wants to simulate a Turing Machine, it should be able to simply take the "simulator" control from the project and use it, without needing the other parts of the project.

   Every part in the program was written as a control which can be compiled to an OCX control. OCX controls can then be used by any other programmers in their projects. A large OCX file can be made for all the controls, or the different controls can be made into separate lightweight OCX files.
7. The ability to convert instructions from pseudo code to Turing Machine instructions

   Even though this aim was not implemented, the theory necessary to implement it was described in section 8 – implementing it is now just a question of time.

### 9.1 – Future work

- Implement all aims left pending above
- Add support for different variations of Turing Machines – doubly infinite models, multi-track models, etc…
- Support a certain degree of reverse-engineering of Turing Machines.
- Include some non-halting detection techniques – for example, if the tape enters the same configuration twice, it will never halt.
- Also allow finite state automatons to be created and simulated by the program.

## 10 – Acknowledgements

First and foremost, my thanks are infinite to my mentor, Bobi Gilburd, who was always there to help me, give me advice and re-assure me, especially when the project was getting hard and going slowly, and who really managed to make the project fun as well as educational. I would never have been able to achieve a quarter of what I did without him. My thanks also go to him as well as my father, Jean Guetta, and Stanislav Tsanev for their proof reading of this report and their invaluable comments.

I would also like to thank the British Technion Society for the very generous help they gave me for my living expenses at the Technion.

And last, but certainly not least, I would like to thank all the SciTech counsellors and participants, who made this month a great, fun and unforgettable experience!

---

**Citations**

The citations below also constitute excellent "further reading" in the subject.

[i] http://babbage.clarku.edu/~djoyce/hilbert/
[ii] Automata and Computability, Dexter C. Kozen – Lecture 1
http://ei.cs.vt.edu/~history/Turing.html
[iii] Introduction to the Theory of Computation, Michael Sipser – Chapter 3
[iv] http://www.ams.org/new-in-math/cover/turing.html
[v] Introduction to the Theory of Computation, Michael Sipser
[vi] http://www.cheransoft.com/vturing/index.html
[vii] For example, http://www.igs.net/~tril/tm/tm.html
[viii] http://www-csli.stanford.edu/hp/Turing1.html
[ix] Visual Basic Algorithms, Rod Stephens
[x] Detailed descriptions of the machines described in the last section of the paper can be found in chapter of 2 of "Introduction to Computability" by Fred Hennie.