



University  
of Glasgow | School of  
Computing Science

Honours Individual Project Dissertation

# TURING MACHINE LANGUAGE

**Pete Gautam**

March 13, 2023

## Abstract

## Acknowledgements

# Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Pete Gautam    Date: 27 September 2022

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation	1
1.2	Objectives	1
1.3	Summary	1
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	Turing Machine	2
2.1.1	Introduction to Turing Machines	2
2.1.2	Executing a TM on a tape	2
2.1.3	TM as a model of computation	4
2.2	Parser	4
2.2.1	Lexical Analysis	4
2.2.2	Syntactic Analysis	5
2.2.3	Semantic Analysis	6
2.2.4	Code Generation	6
<b>3</b>	<b>Requirements</b>	<b>7</b>
3.1	Developing TML	7
3.2	Developing the parser for TML	7
3.3	The Product	8
<b>4</b>	<b>Design</b>	<b>9</b>
4.1	Language	9
4.2	Parser	11
4.2.1	Lexical Analysis	12
4.2.2	Syntactic Analysis	12
4.2.3	Semantic Analysis	13
4.2.4	TM Generation	13
4.2.5	TML Execution	13
4.3	Product	13
4.3.1	TM Conversion	13
4.3.2	Tape Execution	14
<b>5</b>	<b>Implementation</b>	<b>16</b>
5.1	Parser	16
5.1.1	Lexical Analysis	16
5.1.2	Syntactic Analysis	16
5.1.3	Semantic Analysis	17

5.1.4	TM Generation	18
5.1.5	Code and TM Execution	19
5.2	Product	19
5.2.1	Editor	19
5.2.2	FSM Generation	20
5.2.3	TM Formal Definition Conversion	21
5.2.4	Tape Execution	21
<b>6</b>	<b>Evaluation</b>	<b>22</b>
6.1	Evaluation Results	23
6.1.1	Parser and Language Evaluation	23
6.1.2	Product Evaluation	25
6.2	Limitations to Evaluation	26
<b>7</b>	<b>Conclusion</b>	<b>27</b>
7.1	Summary	27
7.2	Reflection	27
7.3	Future Work	27
7.3.1	Language	27
7.3.2	Product	28
	<b>Appendices</b>	<b>30</b>
<b>A</b>	<b>TML Specification</b>	<b>30</b>
<b>B</b>	<b>Proof of Equivalence</b>	<b>34</b>
B.1	Complete TML Programs	34
B.2	Equivalence of TMs and TMLs	37
B.3	TML as a model of computation	41
<b>C</b>	<b>Product Screenshots</b>	<b>43</b>
C.1	Homepage	43
C.2	Documentation Pages	45
C.3	Program Error Pages	46
<b>D</b>	<b>Evaluation Content</b>	<b>47</b>
D.1	Worksheet	47
D.1.1	Introduction to Turing Machine Language	47
D.1.2	Identifying TML Programs	49
D.1.3	Identifying TMs	51
D.1.4	Writing TML Programs	52
D.2	Checklist	52
D.3	Survey	52
	<b>Bibliography</b>	<b>62</b>

# 1 | Introduction

## 1.1 Motivation

Turing Machines are a model of computation that is typically taught to computing students after some years of coding experience. They are normally taught the concept using finite state machines. This is quite different to how they have been taught programming previously, and they might find it easier to learn the concept in a way that resembles coding closely. This project presents a language to represent TMs.

## 1.2 Objectives

There are 3 parts to the project:

1. defining the language,
2. creating the parser for the language and
3. creating a website that allows a user to make use of the parser.

The first aim is to define a programming language to represent Turing machines, called the Turing Machine Language. A Turing machine can be executed on a tape, and this is what the language will simulate.

Next, a parser will be created for the language. This parser should be able to take in a string representation of a program, and then parse it into a program context. Then, a program context can be:

- validated to ensure it has no errors;
- converted into a Turing machine; and
- executed on a tape.

Finally, a website will be created to allow the user to access the parser. It should feature an editor to allow users to type a program. The user would then be able to convert it into a Turing machine, or execute it on a valid tape.

## 1.3 Summary

- **Chapter 2** contains background information on Turing Machines and the parsing process;
- **Chapter 3** lists the requirements for the project;
- **Chapter 4** illustrates the design of the language, the parser and the product;
- **Chapter 5** demonstrates the implementation of the parser and the product;
- **Chapter 6** outlines the results of the evaluation, along with some limitations to the process; and
- **Chapter 7** concludes the dissertation with a summary and highlights some recommendations for future work.

## 2 | Background

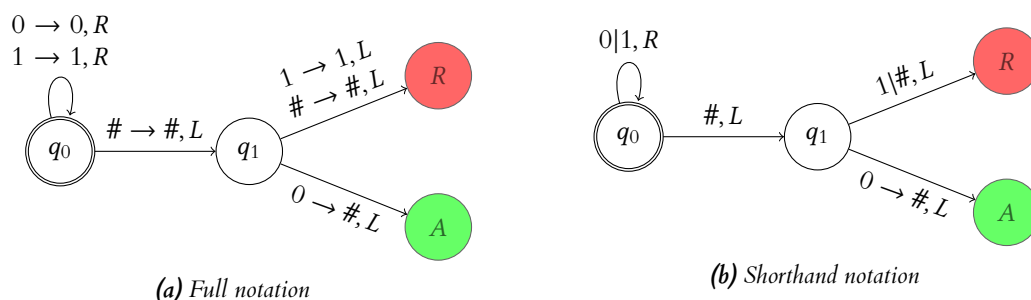
### 2.1 Turing Machine

#### 2.1.1 Introduction to Turing Machines

A **Turing Machine** (TM) is a collection  $(Q, \Sigma, \delta, q_0)$ , where:

- $Q$  is a set of **states**, including the **accept state**  $A$  and **reject state**  $R$ ;
- $\Sigma$  is the set of **letters**, which does not include the **blank** symbol;
- $\delta: Q \setminus \{A, R\} \times \Sigma^+ \rightarrow Q \times \Sigma^+ \times \{\text{left}, \text{right}\}$ , where  $\Sigma^+ = \Sigma \cup \{\text{blank}\}$ , is the **transition function**; and
- $q_0 \in Q$  is the **starting state**.

Although based on Turing's work on Turing (1936), this definition, along with others in this section, have been adapted from Hopcroft et al. (2001).



**Figure 2.1:** A FSM representation of a TM that accepts binary numbers divisible by 2.

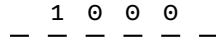
We can represent a TM as a **finite state machine** (FSM). This is a directed graph, with vertices as states and edges as transitions. An example is given in Figure 2.1. In this case, the alphabet  $\Sigma = \{0, 1\}$ . The **blank** symbol is denoted by  $\#$ . The initial state is denoted by  $q_0$ ; the accept state  $A$  and the reject state  $R$ . Every edge corresponds to an evaluation of the transition function  $\delta$ , e.g.  $\delta(q_1, 0) = (A, \text{blank}, \text{left})$ .

The figure presents two ways of representing a FSM- subfigure (a) shows all the transitions, while subfigure (b) only shows the transitions where the tapehead value is getting changed. It also combines letters with similar transitions. We will make use of the shorthand notation.

#### 2.1.2 Executing a TM on a tape

Let  $\Sigma$  be an alphabet. A **tape**  $T$  on  $\Sigma$  is a function  $T: \mathbb{Z} \rightarrow \Sigma^+$ . That is, the tape has infinite entries in both directions. Moreover, each tape entry contains a value from the alphabet, or the **blank** symbol.





**Figure 2.2:** A TM tape on  $\{0, 1\}$ .

We can represent a tape using a figure. For instance, let  $\Sigma = \{0, 1\}$ , and let  $T$  be the tape on  $\Sigma$  given as follows:

$$T(x) = \begin{cases} 0 & x \in \{0, 2, 3\} \\ 1 & x \in \{1\} \\ \text{blank} & \text{otherwise.} \end{cases}$$

Then, Figure 2.2 represents the tape  $T$ . We will assume that the first non-blank value is at index 0.

We can execute a TM on a tape. Let  $M$  be a TM with alphabet  $\Sigma$ , and let  $T$  be a tape on  $\Sigma$ . We execute  $M$  on  $T$  inductively, as follows:

- At any point during execution, we maintain 3 objects: a tape on  $\Sigma$ ; a (current) state in  $M$ ; and an index in the tape (called the **tapehead index**).
- At the start, the tape is  $T$ ; the tapehead index is 0; and the current state is the initial state  $q_0$ .
- At some point during the execution, assume that we have the tape  $S$ , tapehead index  $j$ , with **tapehead value**  $T(j) = t$ , and a non-terminating state  $q$  (i.e. not  $A$  or  $R$ ). Denote  $\delta(q, t) = (q', t', \text{dir})$ . Then,
  - the next state is  $q'$ ;
  - the next tape is  $S'$ , where

$$S'(x) = \begin{cases} t' & x = i \\ S(x) & \text{otherwise;} \end{cases}$$

and

- the next tapehead index is  $j'$ , where

$$j' = \begin{cases} j + 1 & \text{dir} = \text{right} \\ j - 1 & \text{dir} = \text{left}. \end{cases}$$

If the state  $q'$  is not a terminating state, then the execution continues with these 3 objects. Otherwise, execution is terminated with terminating state  $q'$ .



**Figure 2.3:** Some of the tape states during execution.

We illustrate this process with the TM in Figure 2.1 with the tape in Figure 2.2:

- Initially, the tape is the given tape;  $q_0$  is the current state; and the tapehead index is 0, with value 1.
- According to the FSM, we have  $\delta(q_0, 1) = (q_0, 1, R)$ . Hence,
  - the tape remains unchanged;
  - $q_0$  is still the current state; and
  - and the tapehead index becomes 1, with value 0.

- The transition for 0 and 1 are the same with respect to  $q_0$ . This means that we keep moving to the right, until we end up at a blank symbol. At that point, the state of the tape is given in Figure 2.3 (a). The arrow points at the tapehead entry. We are still at the state  $q_0$ , and the tape has not been altered.
- Now, since the tapehead value is `blank`, we move to the left and the current state becomes  $q_1$ . The tape has still not been changed. The current value is now 0.
- We have  $\delta(q_1, 0) = (A, \text{blank}, L)$ . So,
  - the tapehead value changes to from 0 to blank;
  - the current state becomes  $A$ ; and
  - the tapehead pointer move to the left, to index 2.

Since  $A$  is a terminating state, execution terminates, with result `accept`. The final tape state is given in Figure 2.3 (b).

So, the TM in Figure 2.1 executes as follows:

- we use the state  $q_0$  to traverse to the first blank symbol (i.e. the end of the string), and then move to the state  $q_1$ ;
- at state  $q_1$ , we accept the string if and only if the current tapehead value is 0

Hence, this TM accepts binary numbers if and only if they are divisible by 2.

### 2.1.3 TM as a model of computation

Turing initially proposed TMs as the ‘correct’ model of computation in Turing (1936). This result is referred to as the **Church–Turing Thesis**. It is a thesis since it is informal in nature; it is just a *belief* that the correct model of computation is the model given by TMs.

In this paper, Turing also showed that TMs and  $\lambda$ -calculus are equivalent. Hence, it follows that  $\lambda$ -calculus is also the correct model of computation. There have been many other models of computations proposed, such as general recursive functions. It is widely regarded that TMs (and all the equivalent models) represent the correct model of computation. This is because many of the originally proposed models of computation turned out to be equivalent (Copeland (2004)).

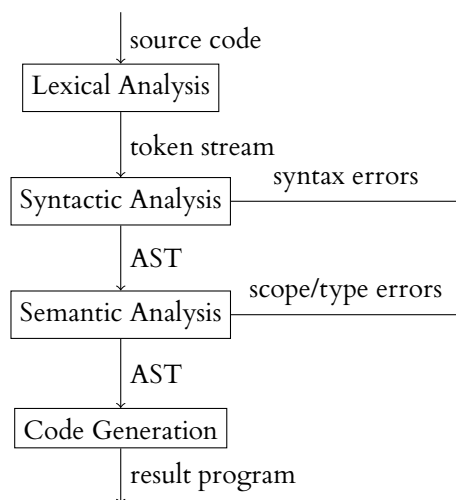
## 2.2 Parser

A **compiler** is a program that takes source code in a programming language (PL) and translates it into a program in another, target, PL. During the process, the compiler also detects any errors, such as syntax and type errors.

We will now consider the different phases of the compilation process. This is summarised in Figure 2.4. This figure, along with most of the content in this section, has been adapted from Aho et al. (2007).

### 2.2.1 Lexical Analysis

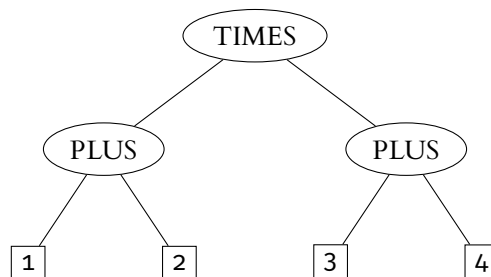
During the compilation process, we first perform **lexical analysis**. In this stage, the source code is enriched to make it ready for parsing. In particular, we generate a stream of source code, which reads the program word by word. Then, it produces a stream of **tokens**. A token is a word in source code along with a label. For instance, consider the mathematical expression `1 + 2`. We can convert this expression into 3 tokens: `(1, NUM)`, `(+, PLUS)` and `(2, NUM)`.



*Figure 2.4: The data flow between the compilation phases.*

### 2.2.2 Syntactic Analysis

Next, we try to parse the token stream into an **abstract syntax tree** (AST). If there are syntax errors present in the program, then it is not possible to construct an AST. This will be detected during the process, at which point we can throw a syntax error.



*Figure 2.5: The AST for the expression  $(1 + 2) * (3 + 4)$*

An AST represents the program as a tree. Typically, the internal nodes represent operations, whereas the leaves represent their arguments. The AST for the expression  $(1 + 2) * (3 + 4)$  is given in Figure 2.5.

There are many ways to parse the stream of tokens. A common method is **recursive-descent parsing**. Here, we have a parser function for each construct in the language, such as a *program*, an *if* command, an *expression*, etc. We look at the next token value and choose the right function.

One way of performing recursive-descent parsing is by **top-down parsing**. In this case, we produce the parent node of the AST and then generate its children.

The simplest form of recursive-descent parsing is called **predictive parsing**. This applies when the next token determines what structure it is to be parsed. For instance, if we see the token *if*, then we know we are parsing an *if* command.

### 2.2.3 Semantic Analysis

Now, we traverse the AST and check that there are no errors in the source code. Typically, there are 2 types of things to check in this stage- **type errors** and **scope errors**.

In type errors, we check whether the AST has some type mismatch, e.g. `1 + true`. We can detect this by keeping track of the types of the identifiers and see if they are legal.

In scope errors, we ensure that all the identifiers present in code are defined. To do so, we need to keep track of all the variables that are in scope. In terms of functions, there is a design choice here- we can have all functions in scope from the start, or add them to scope as they are encountered. Another thing to consider is recursion- to allow recursion, the function must be in scope as soon as it is declared.

### 2.2.4 Code Generation

Finally, we convert the AST into code in the target language. In particular, we traverse the tree and convert each phrase from the source language to the target.

## 3 | Requirements

MoScOWs were used to specify the requirements for the project. In particular, the requirements were partitioned into one of the 4 levels of priority:

- **must have**– this feature is required to construct the minimum viable product;
- **should have**– this feature is required for the product to be practically useful;
- **could have**– this feature is a stretch goal but is plausible; and
- **will not have**– this feature is not something that can be implemented in the given time (or conflicts with another feature).

Since the project has 3 distinct aspects. For this reason, each part had its own MoScOW section. Both functional and non-functional requirements are given.

### 3.1 Developing TML

**Must Have** A specification document for the TML must be created. The specification should include the following:

- a formal and an informal definition for the language; and
- how to execute a program on a valid tape.

Along with the specification, a proof of equivalence between TMs and TML programs should also be provided.

**Should Have** The specification should include examples. In particular, there should be examples of valid and invalid programs, and those that illustrate the proofs (e.g. how to convert a TM into a TML program) so that it is easier to follow. The language should resemble a traditional programming language.

**Could Have** The specification could connect TML program with the Church-Turing Thesis. In particular, a proof of equivalence could be explored between TML program and  $\lambda$ -calculus.

### 3.2 Developing the parser for TML

**Must Have** The parser must be able to:

- parse a string representation of a TM program to a program context;
- validate a program context; and
- execute a program context on a valid tape.

Moreover, the parser must support web deployment and be correct.

**Should Have** The parser should be able to convert a program context to a TM. *Compared to the 3 must-have requirements, this requirement was considered to be of the lowest priority, and so was considered a should-have.*

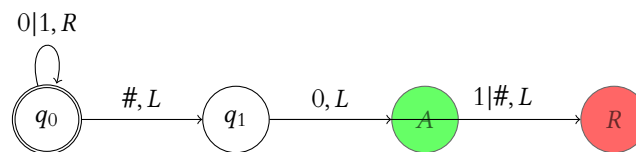
**Could Have** The parser should be able to execute a TM on a tape. This might help in the website to illustrate execution on the converted TM.

**Will Not Have** The parser will not be able to convert a TM into a TML program.

### 3.3 The Product

**Must Have** The website must:

- have a code editor for TML;
- be able to convert a valid program to a TM and present it as a FSM;
- be able to execute a program on a valid tape, one step at a time.



*Figure 3.1: A possible initial rendering of a FSM*

**Should Have** The code editor should support syntax highlighting. Assuming that the website does not make use of any fancy FSM assignment algorithm, i.e. it would produce an initial rendering of FSM such as the one in Figure 3.1, the user should be able to drag states within the FSM to place them in a better position. The website should be fast and easy to use.

**Could Have** The editor could support error detection. The user could be able to configure the website, e.g. change the editor theme, the editor font size and the speed of tape execution. The website could convert a program to its definition as a TM. The website could support automatic placement of states (within the FSM) in an aesthetic manner instead of having the user drag it.

**Will Not Have** The editor will not be able to automatically fix errors. The website will not be able to execute a TM on a tape (without a program). The website will not be able to convert a TM into a TML program.

## 4 | Design

### 4.1 Language

The TML has been designed in a way that closely resembles the operations in a TM. In particular,

- it expects an alphabet like a TM;
- it makes use of `move` commands to move the tapehead pointer in some direction;
- it makes use of `changeto` commands to change the tapehead value to some letter in the alphabet.

Instead of states, the TML has modules. A module can be thought of as a state, although a module is more expressible than a state. We want modules to also be thought of as functions or methods in a traditional PL. To allow for flow of code to go from one module to another, we can make use of `goto` commands. We can go to the *accept* and *reject* states using the keywords `accept` and `reject` respectively.

The following program illustrates a simple program in TML with all the basic operations:

```

1  alphabet = {a, b}
2  module first {
3      changeto blank
4      move right
5      goto second
6  }
7  module second {
8      move left
9      accept
10 }
```

A program is run starting from the first module. In this case, we first start at module `first`. Here, the first tape value is removed, the tape pointer moves to the right and we go to module `b` and continue execution. Note that we allow recursion- line 5 can be replaced with `goto first`.

To represent the transition function in TMs, the language makes use of pattern-matching. Since this should resemble a traditional PL, this is done using `if` cases. This is shown in the example below.

```

1  if a {
2      move right
3      accept
4  } if b, blank {
5      changeto blank
6  }
```

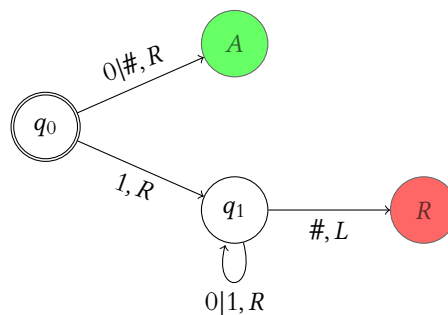
Although the language is already equivalent to TMs, we will add some more flexibility to the language since all programs that can be written with these constructs are quite similar to TMs. To mitigate this, we add nesting within `if` statements. That way, we can write programs that are more comparable to normal programs written in other languages, such as the program below.

```

1 // checks whether a binary number is divisible by 2, recursively
2 alphabet = {0, 1}
3 module isDiv2Rec {
4     // recursive case: not at the end => move closer to the end
5     if 0, 1 {
6         move right
7         goto isDiv2Rec
8     }
9     // base case: at the end => check final letter 0
10    if blank {
11        move left
12        if 0 {
13            accept
14        } if 1, blank {
15            reject
16        }
17    }
18 }

```

In this program, we have nested an `if` block within an `if` block in lines 11–15.



*Figure 4.1: A TM with a self-loop at the state  $q_1$*

Although nesting has made the language more like a typical PL, there is still one issue— a self-loop in a non-starting state. To see this, we consider the TM at Figure 4.1. Currently, the following is the only way to represent this TM as a TML program:

```

1 alphabet = {0, 1}
2 module q0 {
3     if 0, blank {
4         move right
5         accept
6     } if 1 {
7         move right
8         goto q1
9     }
10 }

```



```

11 module q1 {
12     if 0, 1 {
13         move right
14         goto q1
15     } if blank {
16         move left
17         reject
18     }
19 }

```

What we have is a **complete program**– there is a one-to-one correspondence between a module and a state. It is not possible to combine the 2 modules– recursion would convert the self-loop of  $q_1$  to a transition from  $q_1$  to  $q_0$ . It is always possible to represent a TM as a complete program, but this representation is very close to a TM. We want there to be another way to represent this program that resembles a PL better.

To allow for self-loops using nesting, we introduce a new construct– a **while** case. This is similar to an **if** case, but after the block is executed, we stay at the same block. Note that this does not necessarily mean that the same case is run. This is precisely a self-loop. We can now convert the TM to a single module:

```

1  alphabet = {0, 1}
2  module program {
3      if 0, blank {
4          move right
5          accept
6      } if 1 {
7          move right
8          while 0, 1 {
9              move right
10             } if blank {
11                 move left
12                 reject
13             }
14         }
15     }

```

The formal syntax of the language is given in the appendix, along with a proof of equivalence between TMs and TML programs in terms of code execution. The proof of equivalence is composed of several proofs, which involve:

- converting a TM into a complete TML program;
- converting a valid TML program into a complete TML program; and
- converting a complete TML program into a TM program.

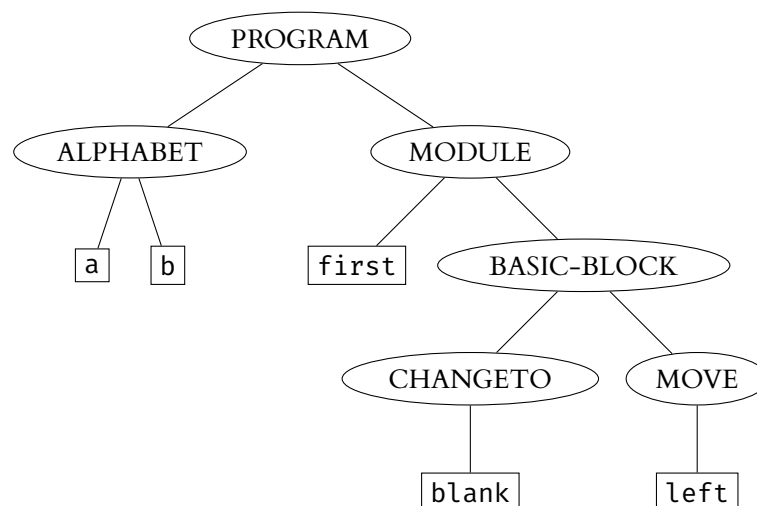
## 4.2 Parser

The parser takes a program in TML and produces a corresponding TM. It also allows for the execution of the TM on a tape. It does so in many steps.

### 4.2.1 Lexical Analysis

The first stage of parsing is lexical analysis, where we produce a stream of tokens from the source code. Since the TML is quite simple, this was decided to be unnecessary- we make use of a stream of source code.

### 4.2.2 Syntactic Analysis



*Figure 4.2: An AST for the TML program with a module called `first`.*

Next, the stream of source code is parsed into an AST. The AST has a node for each command. We illustrate this process with an example. For instance, assume we have the following source code.

```

1 alphabet={a, b}
2 module first {
3     changeto blank
4     move left
5 }
  
```

Then, the parsing process results in the construction of the AST given in Figure 4.2.

The parser is top-down recursive-descent in nature. In particular, when parsing the program above, we try to construct the AST from the root and then fill out the branches and the leaves. In particular, for the program above:

1. We first parse it as a program and construct the root node of the AST;
2. We detect the alphabet at line 1, so we construct the alphabet branch in the AST; and
3. We parse the module `first` from line 2- we construct the module branch and parse all its content.

If successful, this process will result in an AST.

If the parser cannot construct an AST, then the program has some syntax error. In that case, the parser throws an error with a clear and succinct message.

### 4.2.3 Semantic Analysis

After the AST has been constructed, semantic analysis is performed. Since the TML is quite simple, it does not have a type system. Hence, there is no need to do type checking. On the other hand, the language makes use of identifiers, e.g. module names. So, we perform scope checking in this stage. This is done by traversing the AST once.

During this phase, we ensure that a `goto` command refers to a module that is already present in code. By design, we allow the module to be defined anywhere within the document. Moreover, we also check that a module is not defined twice, and is not called `accept` or `reject`.

We also check that the letter of a `changeto` command is one of the letters in the `alphabet` or `blank`. Moreover, there is also a check to validate that a `switch` block contains precisely one case for each letter in the alphabet, i.e. there are no duplicate cases and the cases check all the letters, including `blank`.

### 4.2.4 TM Generation

Next, the AST is used to generate a TM. There are many choices to represent a TM- the formal definition of TMs or FSM representation. To allow for more flexibility during code execution, the formal definition of TMs was chosen.

### 4.2.5 TML Execution

The AST is also used for executing a TML program on a tape. Since a TML program is compiled to a TM, this stage could have been avoided- we could make use of executing TM on a tape. However, this was also included since the execution of a TML program was thought to be more efficient, since it abstracts many of the technicalities presented by TMs. For example, in TMs, each letter in a state should have a different transition, whereas TML supports the same transition for every letter.

## 4.3 Product

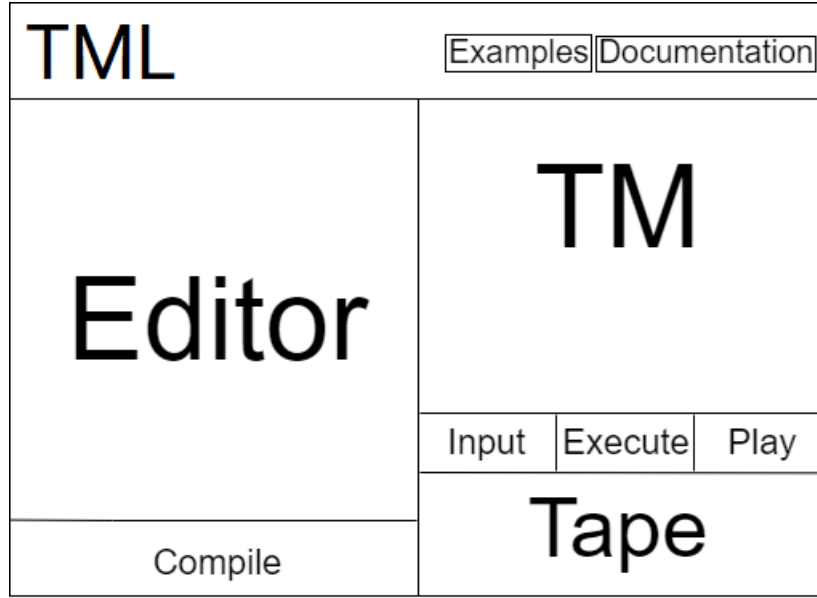
The website was planned to have multiple pages, which included:

- the **homepage** that allowed the user to make use of the parser;
- the **documentation** pages that explained TM and the TML; and
- some pages to illustrate syntax and validation (non-syntax) **errors**.

A prototype of the homepage was created to plan its layout. This is given in Figure 4.3. The editor is placed on the left; the TM on the upper half of the right section; and the tape execution on bottom right. The toolbar features a button to go to the documentation, along with the ability to fill the editor with some example code. The editor can be compiled to a TM, and executed on a tape after the user types an input value. The execute button performs one step in execution, while the play button continually executes the program until it terminates. Note that this does not completely match the implementation.

### 4.3.1 TM Conversion

The website allows live conversion of a (valid) TML program into a TM. This includes constructing the FSM representation of a TM from the program. This is the main conversion since it is quite easy to follow the diagram during execution. To help the user do so, the current state/transition is also highlighted in the FSM.



*Figure 4.3: A prototype for the website homepage.*

	0	1	#
$q_0$	$(R, 0, q_0)$	$(R, 1, q_0)$	$(L, \#, q_1)$
$q_1$	$(L, 0, q_A)$	$(L, 1, q_R)$	$(L, 1, q_R)$

*Table 4.1: A transition table.*

The website also allows the user to convert a TML program into a formal definition of a TM program. The result defines all the states and then presents a transition table, such as the one in Table 4.1. In this example, the non-terminating states are  $q_0$  and  $q_1$ , and the alphabet  $\Sigma = \{0, 1\}$ . Moreover, the transition table says that  $\delta(q_0, 0) = (R, 0, q_0)$ , meaning that, during execution:

- the tapehead pointer moves to the **right**;
- the tapehead value stays **0**; and
- the current state remains  **$q_0$**

### 4.3.2 Tape Execution

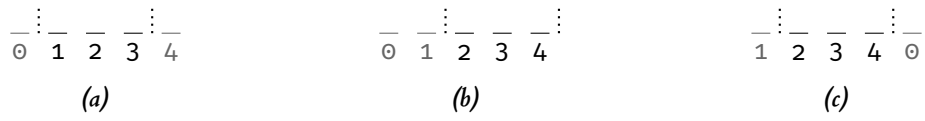
The user can input a valid string in the alphabet and execute the TML program on a tape. The tape panel feature 15 visible tape entries (spots for an input value). The tape panel animates the execution process in the tape entries, which involves:

- changing the tapehead value; and
- moving the tape to the left or the right.

Note that a *move* command moves the *pointer*, not the tape. Hence, the command `move left` moves the *tape* to the right; the tapehead position remains constant.

For the tape movement animation to look smooth, there are always 2 tape entries to either side of the tape. That way, if the tape gets moved to left or right, there will be a tape entry to show. This is illustrated in Figure 4.4 (a)- the tape entries 0 and 4 are out of frame and therefore invisible.

When the tape moves, there will be 2 tape entries on one side. For instance, if the tape moves to the left in Figure 4.4 (a), then we get Figure 4.4 (b). After the transition has completed, we



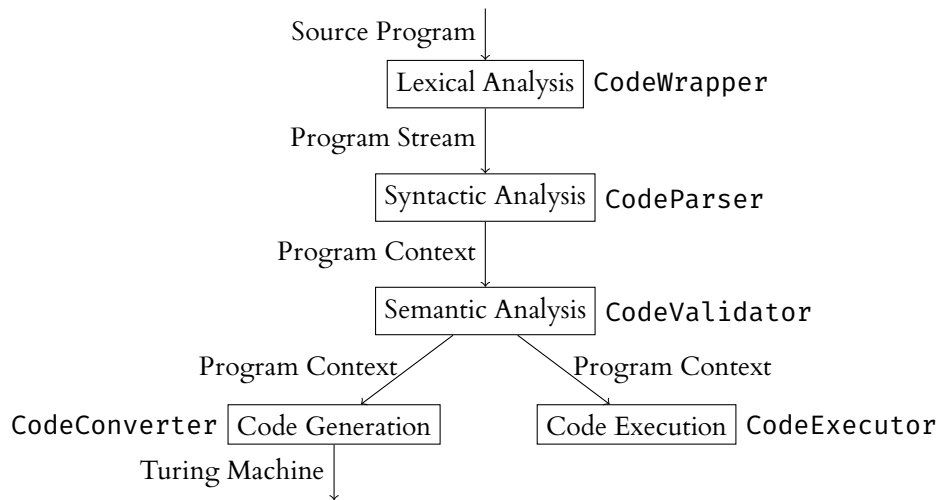
**Figure 4.4:** The transition process for tape entries.

instantly move the most extreme tape entry to the other side. In the example, we move tape entry 0 to the right, leading to the tape state given in Figure 4.4 (c). At this point, we also change the tape value of entry 0 so that it matches tape value at index 5.

## 5 | Implementation

### 5.1 Parser

The parser was written in TypeScript. Although there are many frameworks that could have been used for lexical and syntactic analysis, these were not chosen. The main reason for this is that the parser was meant to be used within a website. In particular, it was expected that the parser become an node package manager (NPM) package. Moreover, TML is quite a simple language, making this task relatively easy and short.



**Figure 5.1:** The parsing process. The process is given inside the box. The class used to achieve the process is given in label, outside of the box. The flow of data is also shown.

The entire parsing process is summarised in Figure 5.1.

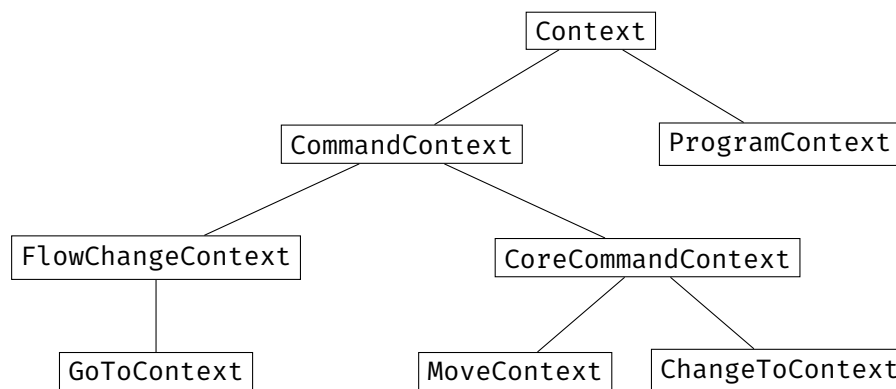
#### 5.1.1 Lexical Analysis

During lexical analysis, a stream of source code was produced. Although the source code was not enriched into tokens, the position of the code was tracked. This was to ensure that, in case of an error, the right section of code could be highlighted. This is done using the class **CodeWrapper**.

To produce a stream of tokens, the iterator design pattern was used. The iterator design pattern allows us to get the current value from a collection in a way that abstracts the data structure (Gamma et al. (1995)). In particular, the pattern was used to abstract the string representation of the source code and return a single entry from the code at a time.

#### 5.1.2 Syntactic Analysis

Next, the code is parsed. This is done using the class **CodeParser**.



*Figure 5.2: A snippet of the Context class hierarchy. All the non-leaf classes are abstract.*

The result of the parsing is a `ProgramContext`, which represents the root of the AST. Subclasses of the `Context` class are used to represent different statements in the class, such as `MoveContext` for `move` commands and `GoToContext` for `goto` commands. A snippet of the class hierarchy for `Context` is shown in Figure 5.2.

The parsing process results in the construction of the AST, such as the one in Figure 4.2. To descend to a child, we make use of the instance fields. For instance, `ProgramContext` has the following fields:

- `alphabet` for `AlphabetContext`; and
- `modules` for an array of `ModuleContext`.

The parser is recursive-descent and top-down in nature, as we can see in the `parseProgram` method below.

```

1  parseProgram() {
2      var alphabet = parseAlphabet();
3
4      var modules = [];
5      while (code.moveNext()) {
6          modules.add(parseModule());
7      }
8
9      return new ProgramContext(position, alphabet, modules);
10 }

```

Note that the code given above is simplified from the actual implementation.

If the program has syntax errors, then it will not be possible to produce an AST. For instance, the alphabet for the program might not been provided. This is detected by the parser since it makes use of predictive parsing. In particular, the parser looks at the next value and determines if it is legal; otherwise, it throws a `SyntaxError`.

### 5.1.3 Semantic Analysis

During semantic analysis, we traverse the AST using the **visitor design pattern**. The visitor design pattern allows us to construct the same method for a class hierarchy without making changes to the classes (Gamma et al. (1995)). In particular, we want to create a method to validate each `Context`.

To allow for visitor design pattern, a concrete `Context` class implements the method `validate`. Each concrete class makes use of the right method from the `CodeValidator` class. We illustrate this with an example. Consider the following snippet of `GoToContext`.

```

1  class GoToContext extends FlowChangeContext {
2      identifier:string;
3
4      validate(validator:CodeValidator):void {
5          return validator.validateGoTo(this);
6      }
7
8      // ... other methods for goto context
9  }

```

In that case, we can define the following method to ‘visit’ a *goto* statement and validate it.

```

1  class TypeValidator {
2      validate(context:Context):void {
3          return context.validate(this);
4      }
5
6      // validate that the goto identifier a module name
7      validateGoTo(gotoContext:GoToContext):void {
8          if (!moduleNames.contains(gotoContext.identifier)) {
9              throw new CodeError("Undefined name- " + identifier);
10         }
11     }
12
13     // ... other validator methods
14 }

```

To run the `CodeValidator`, we validate `ProgramContext`.

When traversing the AST, we typically want to aggregate the result or share the result with the parent. For this reason, we typically return a specific type of values within each visit method.

In `CodeValidator`, we return `true` if a block has a flow command. This data is used in containers that have multiple blocks, such as a module. We throw an error if a non-final block has a flow command. Note that we always return `true`, e.g. even in `ProgramContext`, since the pattern must be followed by all the subclasses of `Context`.

The advantage of using the visitor design pattern is that we have a way of traversing the AST without altering any of the `Context` classes; we can just construct a `Visitor` class. However, if we wanted to add another `Context` subclass, then we would need to amend all the `Visitor` classes. We expect the language to be pretty static, so this is perfectly fine in our case.

#### 5.1.4 TM Generation

Next, we convert the AST into a TM. The TM is implemented in the class `TuringMachine` and mimics the definition of a TM.

We traverse the AST using the visitor design pattern. Initially, we define the TM and add relevant states and transitions during the traversal. The visitor methods return the label of the next state,



if applicable to the context. This is defined in a very complex manner, depending on whether we have an *if* or a *while* command, or none at all.

### 5.1.5 Code and TM Execution

Finally, a validated program can be executed using `CodeExecutor`. This class follows the iterator design pattern. In particular, we can make use of the method `execute` to run one step in execution. This method returns `false` if and only if execution has not terminated. The steps of execution are defined precisely in the language specification, given in the appendix.

Unlike the previous two stages, code execution does not make use of the visitor design pattern. This is because we do not need to traverse the AST in one go to convert the program. Instead, it makes more sense to use the iterator design pattern– this supports execution one step at a time.

TM execution has also been implemented in the class `TMExecutor`. This also makes use of the iterator design pattern and supports execution one step at a time.

## 5.2 Product

Due to the complex nature of the website, it made use of many APIs.

At the start of the implementation, 3 frameworks were considered for the website:

- the **Webpack** framework– it has little overhead, allows for a lot of flexibility, but does not directly have components (such as toolbar and drawer) or support state management;
- the **React** framework– it has a rich collection of components and supports state management directly;
- the **Angular** framework– like React, it has a rich collection of components and supports state management.

**State management** was an important consideration when choosing the framework since the website tracks many states, such as the value of the editor and the current TM. Hence, it is particularly important that state management is readily supported by the framework. For this reason, webpack was not chosen.

Both React and Angular would have been equally good choices for the project. React was chosen as there are more APIs that readily integrate with React compared to Angular, includes some of the APIs used in this project. This is true since React is more widely used than Angular. The React framework supports coding in either JavaScript or TypeScript. The project made use of the TypeScript version for type safety.

The website made use of **Material Design**. The components provided by the API are common-purpose, such as tables and drawers, and are easy to configure. Moreover, Material design is quite widespread since all Google products make use of it. Hence, the user is expected to recognise these common constructs and should be able to easily interact with them. For instance, the user can recognise that a settings icon allows them to customise the website in some way. Furthermore, Material Design helps keep the website consistent and responsive.

Screenshots of the website homepage, documentation and error pages are given in the appendix.

### 5.2.1 Editor

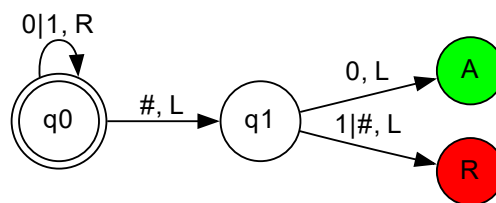
The editor feature was implemented using the **monaco** API. The API was chosen since it integrates well with React and provides many features. The features include:

- syntax highlighting with different priorities (e.g. an error gets an error priority while a code execution highlighting gets a low priority);

- the ability to easily set and get the current value; along with
- numerous customisations to the editor, such as as changing the font size, setting an editor theme and showing/hiding line numbers- these are all features that a user can configure in the website.

Moreover, the monaco editor was chosen since it has the same look as Visual Studio (VS) Code, and shares many functionalities with the IDE. A developer survey in 2022 found that VS Code is the most popular code editor among 70 000 developers.

### 5.2.2 FSM Generation



*Figure 5.3: The graphviz rendering of the directed graph `isDiv2`.*

The FSM representation of the TM is created using the **graphviz** framework. This makes use of the **DOT** notation, which represents graphs in text by listing their nodes and edges. In particular, a graph in DOT notation is given below:

```

1 digraph isDiv2 {
2   node [shape = "doublecircle"]; q0;
3   node [shape = "circle"]; q1;
4   node [style = "filled", fillcolor = "green"]; A;
5   node [style = "filled", fillcolor = "red"]; R;
6   q0 -> q0 [label = "0|1, R"];
7   q0 -> q1 [label = "#, L"];
8   q1 -> A [label = "0, L"];
9   q1 -> R [label = "1|#, L"];
10 }

```

This graph represents the FSM representation of a TM. The keyword **digraph** implies that the graph is directed. The keyword **node** is used to create states. We use an arrow symbol (**->**) to construct edges between states.

Using the graphviz API, the graph in DOT notation can be rendered in the website- the result for the directed graph `isDiv2` is given in Figure 5.3. Note that the figure was produced with some extra formatting code that is not shown.

We convert a TM to DOT notation as follows:

- we set the default formatting of a node to be a circle (similar to code in line 3 of the example);
- we list the initial state `q0`, the accept state `A` and the reject state `R`, with the right formatting (like in lines 2, 4 and 5); and

- we then list all the edges and label it with the corresponding transition (like in lines 6-9).

The advantage of using the `graphviz` package is that it can produce a very well-formatted FSM. Initially, the graph was rendered by placing the states in order of its label, like in Figure 4.1. The user would be able to drag the states and hence achieve a more reasonable placement. However, there was time to make use of the `graphviz` package later in the project, and so the implementation was changed. Now, the user is unable to drag the states, but this should not be necessary as the states are already well-placed!

### 5.2.3 TM Formal Definition Conversion

The formal definition of a TM is constructed quite naturally by using the class `TuringMachine`—we can look at the `transition` function to find an object of type `TMChange`, which shows:

- the next state value;
- the direction to move; and
- the value the tapehead value will become.

These 3 values are shown to the user in a table like the one in Figure 4.1.

The text in the table is constructed using the **MathJax** framework. This framework compiles raw code in LaTeX and renders it as an SVG.

### 5.2.4 Tape Execution

The tape was implemented as a SVG elements. These were constructed using the **d3** API. To illustrate tape execution, we make use of the animation subpackage in **d3**. In particular, this is used to move the tape left or right, and to change its value.

During execution, we also highlight the current block being executed. Moreover, if the TM panel has been rendered, we highlight the current state and transition in the FSM representation. To keep track of these values, we execute the tape on both the TML program (using `CodeExecutor`) and its corresponding TM (using `TMExecutor`). We make use of the **d3** API here as well to animate the change in current block/state.

## 6 | Evaluation

The project had 2 aspects to evaluate- the TML and the product (website). Both aspects were evaluated continually through unit tests during production and tested using a user evaluation.

After the product had been completed, a user evaluation was conducted on 18 second year computing students. TMs are taught to students that have few years of programming experience, and for this reason second years were chosen. They had little familiarity, if any, with Turing Machines and were introduced to both TMs and TML during the evaluation session.

The aim of the evaluation was for them to get acquainted with TMs and TML programs, and then to:

- compare TMs and TML programs;
- understand whether writing a TML program would help drawing a TM; and
- evaluate the product.

The evaluation session also served as a great opportunity to ask for any features to be added to the product and the language.

During the evaluation session, students were first introduced to TML programs. They were then expected to understand what language two mystery TML programs accept. They were expected to do so by checking whether the programs accepted some values. During this process, they should have been able understand the way the program operates and decode the language it accepts. They were expected to use the product to help them follow the code and understand the steps in execution.

The students were then introduced to TMs, and expected to decode 2 TMs in a similar manner. Since the website can only execute TML programs, they were given TML programs for the TM. Note however that this did not defeat the purpose of testing TMs since the programs given were complete TML programs, which is essentially another representation for TMs.

Finally, they were asked to write some programs in the TML. Like in the previous sections, they were free to use the website to write the programs and test its correctness. The first few programs were quite similar to the code they had seen before. The remaining programs were somewhat more difficult, and for this reason they were optional. Nonetheless, many students attempted them and wrote impressive programs!

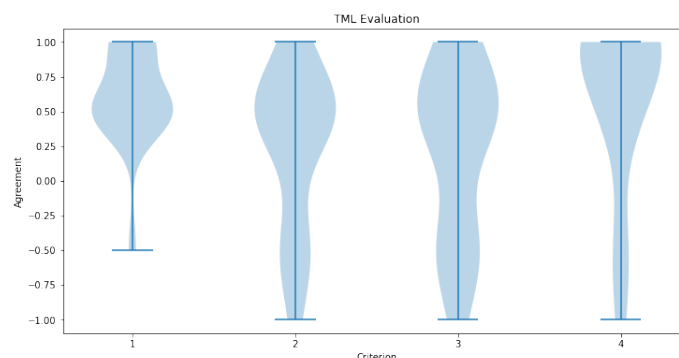
The evaluation took about 50 minutes to be completed. The students were asked to fill out a worksheet with their answers.

After they had completed the worksheet, they completed a survey to evaluate the language and the product. To avoid writing programs on paper, students were asked to copy their code from the final part of the worksheet to the survey. The results of the survey are discussed in detail in the next section. Both the worksheet and the survey are given in the appendix.

## 6.1 Evaluation Results

### 6.1.1 Parser and Language Evaluation

The parser was continually tested during production to ensure correctness. This was achieved using unit testing. They were used to test all aspects of the parser and were extensive. In fact, the unit tests had more than 95% code coverage.



**Figure 6.1:** A violin plot that summarises the results of the survey relating to the TML with respect to the 4 criteria. The agreement value refers to how much the user agrees to the statement:  $-1$  is strongly disagree;  $-0.5$  disagree;  $0.5$  agree and  $1$  strongly agree. The whiskers show the range of answers, e.g. nobody said they strongly disagreed with criterion 1. Moreover, the density is proportional to the number of participants answering the question with that value, e.g. most people answered criterion 1 with the value  $0.5$  (agree).

During user evaluation, users were asked to evaluate the language in the following criteria:

1. TML is easy to understand
2. TML is easy to write programs in
3. I was able to fix errors in my code using the error messages provided
4. I was able to easily reason executing a program on a tape

They were asked to rate how much they agreed with each statement, and the result is summarised for each criterion in Figure 6.1.

It is clear that most students found the language easy to understand. Students noted that the syntax is quite similar to Java, a language they are quite familiar with. It is also clear from the worksheet solutions that the language is easy to follow- most students were able to correctly identify and explain which values a TML program accepts.

However, a smaller number of students believed that the language is easy to write programs in- the solutions to the worksheet illustrate that some students found it harder to write (syntactically) correct programs than to understand it. This is expected given that they were only exposed to the language for about an hour.

Many found the error messages quite useful and it helped them write correct programs, but few disagreed with this. A student mentioned that some of the error messages could have been given more details, for example a missing case error did not identify what letter in the case was missing. This issue was fixed after the evaluation sessions- now, the missing letter is mentioned in the error message.

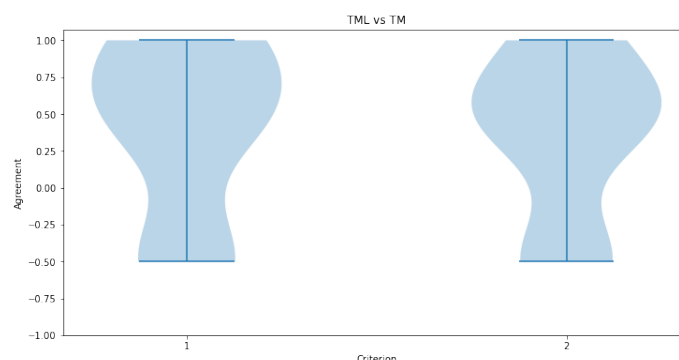
Most students found it easy to reason executing a program on a tape- they were able to run the code on the website and follow the code quite easily. Since the students were not formally introduced to the language, some of the students struggled to reason execution of the code they had written. For instance, consider the following block of code:

```

1  if a {
2    changeto blank
3    goto someModule
4  }

```

When this code is executed, and the tapehead value is `a`, the value gets changed to `blank` as expected. However, the tape also moves to the `left` before moving to the module `someModule`. This is the default behaviour, but the students were not informed of this. If there was more time for evaluation, the rules of execution would have been explained more thoroughly. Nonetheless, by asking the students to explicitly include the commands, this issue was partially mitigated.



**Figure 6.2:** A violin plot that summarises the results of the survey that compare TM and TML with respect to the 2 criteria.

The TML was then directly compared to TMs. In particular, students were asked to compare TMs and TML programs in the following criteria:

1. I am more confident in writing a program in TML than drawing a TM
2. I find it easier to reason what a TML program accepts than a TM

The students were asked to rate how much they agreed with each statement, and the result is summarised for the two criteria in Figure 6.2.

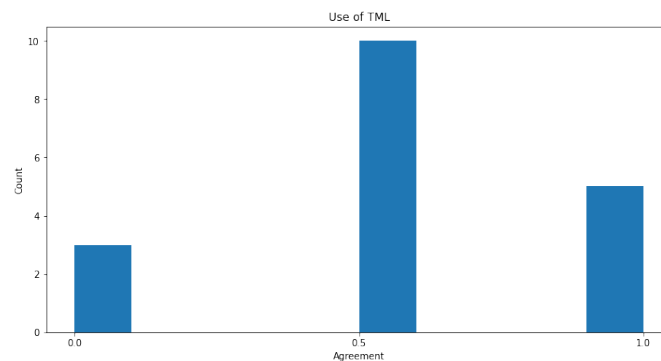
Overall, students seem to be more comfortable with TML than TM. They were more confident writing a TML than drawing a TM– this is expected since they were not asked to draw a TM. From the worksheet, it is evident that students were able to replicate the code given and write correct programs, and to a lesser extent, devise new programs.

It is somewhat surprising that many found it easier to reason a TML program than a TM. This is because many claimed that it is easier to follow the TM diagram than code. Nonetheless, the students might have found it easier to reason a TML program since the worksheet focuses much more on TML than TM.

The students were then asked whether they would consider writing a TML program before drawing a TM. The response of this question is summarised in Figure 6.3.

When drawing a TM for some algorithm, it is quite helpful to plan the machine beforehand. TML provides an opportunity where it is possible to reason in quite low-level how the algorithm is meant to execute on a tape without considering the states and transitions in a TM. Moreover, it is quite easy to convert a TML to a TM. I believe this is the main selling point of the language.

From the results, it is clear that many would consider drawing it. The hesitation might result from the little experience that they had gotten. Moreover, most students only attempted questions



**Figure 6.3:** A histogram that summarises whether the users would consider writing a TML program before drawing a TM. 0 means no; 0.5 means maybe; and 1 means yes.

involving simple algorithms. Nonetheless, the few students that attempted the harder questions exclaimed that they would struggle directly drawing a TM for those algorithms.

In summary, the TML language seems to be a promising alternative to TMs. Due to the short length and small number of participants of the evaluation, it cannot be concluded whether it is easier to learn about TMs using FMs or through TML programs. Nonetheless, it seems that TML programs are relatively easy to learn and understand. Further evaluation would help compare it more thoroughly with TMs and explore the true potential of the language.

### 6.1.2 Product Evaluation

Like with the language, the product was also continually tested during production for correctness. This was achieved through unit testing. Unlike the testing for language, this was however less successful due to the limits in mocking frameworks and time constraints. Nonetheless, the tests covered all the major parts of the website and ensured that all the functionalities implemented were correct. If there was more time, the mocking would be more thorough so that the tests could be exhaustive.

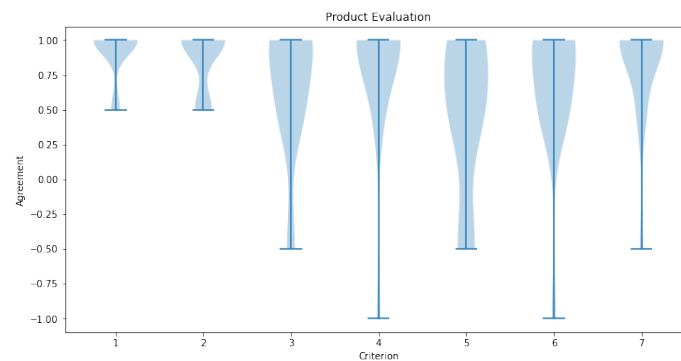
During the user evaluation, the students were expected to use the product to understand TMLs and TMs. For this reason, they were able to evaluate the website. In the survey, they were asked to evaluate the product in the following 7 criteria:

1. The website is easy to follow
2. The presentation of the website is intuitive
3. There were no visible bugs in the website
4. The website was fast
5. The website feels complete
6. The code execution was easy to follow
7. The code editor was easy to use

The results of the survey are summarised in Figure 6.4.

It is evident that most found the website easy to follow and intuitive to use. Unfortunately, there were a few bugs present, e.g. the TM did not change to the latest version when tape execution began. These bugs were fixed later after the evaluation sessions. Most also found the website to be fast and complete.

Some did not find code execution easy to follow- this was particularly the case with long programs where students had to scroll to find the currently executing block of code. Also, some found the



**Figure 6.4:** A violin plot that summarises the results of the survey relating to the website with respect to the 7 criteria.

code editor difficult to use. There were some issues in using the code editor- the code cannot be edited while being executed, but there was no feedback to inform the user about this issue. A pop-up has now been added to warn them of this issue.

Overall, it is clear that the website serves as a good platform to parse a TML program and execute it on a tape. There are still some issues with the website that can be fixed later. Nonetheless, the website feels complete and functions as expected.

## 6.2 Limitations to Evaluation

Due to the time constraints of the project, there were some limitations to the evaluation, in particular the user evaluation. The biggest limitation was the length of the evaluation session. It was hard to conduct a productive, short and accessible session.

For this reason, it was not possible to test the students' ability to draw TMs. Moreover, a significant portion of the question do not require the student to understand TMs or TML programs; they can just run the code on the website and get the answer. This was done to help the students grasp the language easily. When it came to describing the values that a program/TM accepted, it was clear that some had not understood the program. For instance, in a mystery program that accepts values that are  $3 \bmod 4$ , some students claimed that the program accepts all odd numbers!

Moreover, the students' ability to write some TM programs could not be tested completely. In fact, the core questions only involved making minor changes to the programs they were given; it was only the optional questions that truly tested their ability to write TM programs and reason about them.



## 7 | Conclusion

### 7.1 Summary

TODO

### 7.2 Reflection

TODO

### 7.3 Future Work

There are many additions that can be made to the language and the product in future. Some of these were discovered during production, and some as part of the user evaluation.

#### 7.3.1 Language

Although the language is equivalent to TMs, and has a few features that resemble a traditional PL, it is still quite low-level. There are many common paradigms that can be added to the language. These include:

1. the ability to traverse to the end (or the start) of the tape string in one command, e.g. `move end`;
2. an *else* block (an *if* block for the remaining letters); and
3. the ability for modules to be parameterised with respect to letters.

We illustrate these issues with the following program:

```

1 // accept strings that start and end in opposite letters
2 alphabet={a, b}
3 module oppositeLetters {
4     if a {
5         move right
6         while a, b {
7             move right
8         } if blank {
9             move left
10            if b {
11                accept
12            } if a, blank {
13                reject
14            }
15        }
16    } if b {
17        move right

```

```

18     while a, b {
19         move right
20     } if blank {
21         move left
22         if a {
23             accept
24         } if b, blank {
25             reject
26         }
27     }
28 }
29 }

```

The first feature is a very common feature found in program- many programs involving some check on the final character. For example, in the program above, we are traversing to the end twice- at lines 5-9 and 17-21. Hence, I believe this would be a highly beneficial feature to add.

The second point is also somewhat common and was suggested in the survey. At many points during execution, we want to do something for a single letter and something different for the other letters. For instance, in the example above, at lines 10-13, we want the program to accept the string if and only if the letter is a **b**. If the alphabet was longer, this would be quite inefficient. Also, another point that was raised during the survey was that the language only makes use of pattern-matching, and could be more flexible. So, this would be a great feature to extend the language.

The final feature is quite interesting. There are many programs where the modules are very similar and only differ in some letters. For example, the program above has 2 essentially mirrored blocks at lines 4-16 and 16-28. In particular, the only difference in these lines of code is at lines 10/12 and 22/24, where the letters **a** and **b** are the other way round. Hence, I believe that adding this feature would make many programs in TML shorter and less repetitive! Moreover, this would make the language further resemble a traditional PL.

### 7.3.2 Product

There are many possible improvements to the website. These are some of the features that can be added:

- support for direct execution of a TM;
- a play button on the tape section to execute long programs on long tapes without pressing the step button many times;
- the ability to collapse the panels; and
- the ability to customise the number of tape entries shown so that it is easier to follow code execution on long strings.

All of these are great features that could be added to the website and would make it easier to use!

We could also improve the website by making the TM panel more responsive. The FSM is currently being produced using the graphviz framework. The resulting SVG has hard-coded dimensions. This makes it hard to make the panel responsive. It is not completely possible to make the graph fixed; only the maximum size of the SVG can be specified. Moreover, the constraints are not taken into consideration when the API produces the FSM. This means that for complex TM, the states are quite small and hard to see. This makes it hard to follow the execution process. To fix this, we could add one of the following features:

- the FSM might produced as it currently is, but the user can zoom in and drag the FSM panel; or

- the FSM might be produced so that the states always have the same size, but the user can scroll the panel.

## A | TML Specification

In this chapter, we will define the syntax of the TML, starting with an example. We next analyse the syntax and define execution of a valid TML program on a tape in a similar manner to the execution of a TM.

Consider the following TML program.

```

1 // checks whether a binary number is divisible by 2
2 alphabet = {0, 1}
3 module isDiv2 {
4     // move to the end
5     while 0, 1 {
6         move right
7     } if blank {
8         move left
9         // check last letter is 0
10        if 0 {
11            accept
12        } if 1, blank {
13            reject
14        }
15    }
16 }
```

A program in TML will be used to execute on a tape, so the syntax used guides us in executing the program on a tape.

- A valid TML **program** is composed of the **alphabet**, followed by one or more **modules**. In the example above, the alphabet of the program is {0, 1}, and the program has a single module called `isEven`.
- A module contains one or more **blocks** (a specific sequence of commands). There are two types of blocks- **basic blocks** and **switch blocks**.
- A basic block consists of **basic commands** (*changeto*, *move* or *flow* command). A basic block consists of at least one basic command, but it is not necessary for a basic block to be composed of all the basic commands. If multiple commands are present in a basic block, they must be in the following order- *changeto*, *move* and *flow* command. In the program above, there are many basic blocks, e.g. at lines 4, 6, 8-9 and 11-12. We do not say that line 8 is a basic block by itself; we want the basic block to be as long as possible.
- A **switch block** consists of cases (*if* or *while* commands), each of which corresponds to one or more letters. A switch block must contain precisely one case for each of the letter in the alphabet, including the `blank` letter. The first block within a case block cannot be another switch block. In the program above, there is a switch block at lines 3-14 and a nested switch block at lines 7-13.

- The body of an *if* command can be composed of multiple blocks. These blocks can be both basic blocks and switch blocks. We can see this at lines 5–13; the *if* block has a basic block at line 6 and then a switch block.
- The body of a *while* command must be composed of a single basic block. The basic block cannot have a *flow* command. This is because when we execute a *while* block, the next block to run is the switch block it is in; we cannot accept, reject or go to another module.
- A switch block must be the final block present; it cannot be followed by a basic block.

```

program = alphabet module+
alphabet = alphabet = { seq-val }
module = module id { block+ }
block = basic-block | switch-block
switch-block = case-block+
case-block = if-block | while-block
if-block = if seq-val { block+ }
while-block = while seq-val { core-com+ }
basic-block = (core-com | flow-com)+
core-com = move direction | changeto value
flow-com = goto id | terminate
terminate = reject | accept
direction = left | right
seq-val = (value,)* value
value = blank | a | b | c | ... | z | 0 | 1 | ... | 9
id = (a | b | c | ... | z | A | B | C | ... | Z)+

```

**Figure A.1:** The EBNF of the TML.

The EBNF of the TML is Figure A.1.

We will now consider how to execute a tape on a valid TML program. Let  $P$  be a TML program with alphabet  $\Sigma$  and let  $T$  be a tape on  $\Sigma$ . We execute  $P$  on  $T$  inductively, as follows:

- At any point during execution, we maintain 3 objects– a tape on  $\Sigma$ , a block of  $P$  and the tapehead index.
- At the start, the tape is  $T$ ; the tapehead index is 0; and the block is the first block in the first module in  $P$ .
- At some point during the execution, assume that we have the tape  $S$ , tapehead index  $j$ , with tapehead value  $T(j) = t$ , and a block  $b$ . We define the next triple as follows:
  - if  $b$  is a *switch* block, we take the first block from the case corresponding to the tapehead value– because the program is valid, this is a basic block; we will now refer to this block as  $b$ .
  - if  $b$  has a *changeto val* command, the next tape  $T'$  is given by

$$T'(x) = \begin{cases} \text{val} & x = i \\ T(x) & \text{otherwise.} \end{cases}$$

If the *changeto* command is missing, then the tapehead  $T' = T$ .

- if  $b$  has a *move dir* command, the next tapehead index is given by:

$$i' = \begin{cases} i+1 & \text{dir} = \text{right} \\ i-1 & \text{dir} = \text{left}. \end{cases}$$

If the *move* command is missing, then  $i' = i - 1$ .

- we either terminate or determine the next block  $b'$  to execute (in decreasing precedence):
  - \* if the block is the body of a while case block, then the next block  $b' = b$ , i.e. we execute this switch block again (not necessarily the same case block);
  - \* if the block contains a terminating *flow* command, execution is terminated and we return the terminated state (*accept* or *reject*);
  - \* if the block contains a *goto mod* command, then  $b'$  is the first block of the module *mod*;
  - \* if the block is not the final block in the current module, then  $b'$  is next block in this module;
  - \* otherwise, execution is terminated and we return the state *reject*.

If execution is not terminated, execution continues with the next triplet.

— 1 0 0 0 —

**Figure A.2:** A TM tape on  $\{0, 1\}$ .

We will now illustrate this process. So, consider the following TML program:

```

1 // checks whether a binary number is divisible by 2
2 alphabet = {0, 1}
3 module isDiv2 {
4   // move to the end
5   while 0, 1 {
6     move right
7   } if blank {
8     move left
9     // check last letter is 0
10    if 0 {
11      accept
12    } if 1, blank {
13      reject
14    }
15  }
16 }
```

We execute this program on the tape at Figure A.2.

- Initially, the tape is the given tape; the current block is at lines 5–15; and the tapehead index is 0, with value 1.
- Since the tapehead value is 1, the basic block to be executed is lines 5–7. Hence,
  - the tape remains unchanged;
  - the current block is still at lines 5–15; and
  - and the tapehead index becomes 1, with value 0.

- The transition for  $\emptyset$  and **1** are the same with respect to the current block. This means that we keep moving to the right until we end up at a blank symbol. At that point, the following is the state of the tape:

$$\begin{array}{ccccccc} & & \mathbf{1} & \emptyset & \emptyset & \emptyset & \\ - & - & - & - & - & - & - \\ & & & & & & \uparrow \end{array}$$

The arrow points at the tapehead value. We are still executing the same block, and the tape has not been altered.

- Now, since the tapehead value is **blank**, we move to the left. Moreover, the current block is at lines 10–14. The tape has still not been changed. The current value is now 0.
- The tapehead value is currently  $\emptyset$ . So,
  - the tape remains unchanged;
  - we have reached the **accept** command; and
  - the tapehead pointer move to the left (by default), to index 2.

Hence, execution terminates, with result **accept** and the following tape state:

$$\begin{array}{ccccccc} & & \mathbf{1} & \emptyset & \emptyset & \emptyset & \\ - & - & - & - & - & - & - \\ & & & \uparrow & & & \end{array}$$

## B | Proof of Equivalence

In this chapter, we give a proof of equivalence between TMs and TML programs. This is done in many steps that involve:

- proving that a TM can be converted to a complete TML program;
- proving that a complete TML program can be converted to a TM; and
- proving that a valid TML program can be converted to a complete TML program.

### B.1 Complete TML Programs

When we defined execution of a valid TML program on a tape in the specification, we said that a basic block need not have all 3 types of commands (*changeto*, *move* and a *flow* command), but in the execution above, we have established some ‘default’ ways in which a program gets executed. In particular,

- if the *changeto* command is missing, we do not change the value of the tape;
- if the *move* command is missing, we move left;
- if the *flow* command is missing, we can establish what to do using the rules described above—this is a bit more complicated than the two commands above.

Nonetheless, it is possible to include these ‘default’ commands to give a **complete** version of the program. This is what we will establish in this section.

Consider the following complete program.

```

1  alphabet = {0, 1}
2  module isOdd {
3      // move to the end
4      while 0 {
5          changeto 0
6          move right
7      } while 1 {
8          changeto 1
9          move right
10     } if blank {
11         changeto blank
12         move left
13         goto isOddCheck
14     }
15 }
16 module isOddCheck {
17     // accept if and only if the value is 1
18     if 0, blank {
19         changeto 0
20         move left
21         reject

```



```

22   } if 1 {
23       changeto blank
24       move left
25       accept
26   }
27 }

```

Now, we consider the rules that a complete TML program obeys:

- A basic block in a complete program has all the necessary commands- if the basic block is inside *while* case, it has a *changeto* command and a *move* command; otherwise, it also has a *flow* command.
- A module in a complete program is composed of a single switch block.

We will now construct a complete TML program for a valid TML program.

1. We first break each module into smaller modules so that every module has just one basic/switch block- we add a *goto* command to the next module if it appeared just below this block.
2. Then, we can convert each basic block to a switch block by just adding a single case that applies to each letter in the alphabet.
3. Finally, we add the default values to each basic block to get a complete TML program.

This way, we can associate every block in the valid program with a corresponding block in the complete program. The complete version is always a switch block and might have more commands than the original block, but it still has all the commands present in the original block.

We now illustrate this process with an example. Assume we first have the following program.

```

1  alphabet = {a, b}
2  module simpleProgram {
3      changeto b
4      move left
5      move right
6      accept
7  }

```

In step 1 of the completion process, we create a module for each block. In this program, there are two basic blocks- at lines 3-4 and 5-6. So, after applying the first step, we get the following program.

```

1  alphabet = {a, b}
2  module simple1 {
3      changeto b
4      move left
5      goto simple2
6  }
7  module simple2 {
8      move right
9      accept
10 }

```

In this case, we have two basic blocks at lines 3-5 and 8-9. So, in step 2, we convert them into switch blocks and get the following program.

```

1  alphabet = {a, b}
2  module simple1 {
3      if a, b, blank {
4          changeto b
5          move left
6          goto simple2
7      }
8  }
9  module simple2 {
10     if a, b, blank {
11         move right
12         accept
13     }
14 }

```

Finally, we add all the default values in step 3 and get the following program.

```

1  alphabet = {a, b}
2  module simple1 {
3      if a {
4          changeto b
5          move left
6          goto simple2
7      } if b {
8          changeto b
9          move left
10         goto simple2
11     } if blank {
12         changeto b
13         move left
14         goto simple2
15     }
16 }
17 module simple2 {
18     if a {
19         changeto a
20         move right
21         accept
22     } if b {
23         changeto b
24         move right
25         accept
26     } if blank {
27         move right
28         accept
29     }
30 }

```

This program obeys the definition of a complete program.

**Theorem 1.** *Let  $P$  be a valid TML program. Then,  $P$  and its completion  $P^+$  execute on every tape  $T$  in the same way. That is,*

- *for every valid index  $n$ , if we have tape  $T_n$ , tapehead index  $i_n$  and module  $m_n$  with executing block*

$b_n$  for the TM program  $P$ , and we have tape  $S_n$ , tapehead index  $j_n$  and module  $t_n$ , then  $T_n = S_n$ ,  $i_n = j_n$ , and  $t_n$  is the corresponding complete module block of  $b_n$ ;

- $P$  terminates execution on  $T$  if and only if  $P^+$  terminates execution on  $T$ , with the same final status (*accept* or *reject*).

*Proof.* We prove this by induction on the execution step (of the tape).

- At the start, we have the same tape  $T$  for both  $P$  and  $P^+$ , with tapehead index 0. Moreover, the corresponding (completed) module of the first block in the first module of  $P$  is the first module of  $P$ . So, the result is true if  $n = 0$ .
- Now, assume that the result is true for some integer  $n$ , where the block  $b_n$  in the TML program  $P$  does not end with a terminating *flow* command. Let  $\sigma_n$  be the letter at index  $i_n = j_n$  on the tape  $S_n = T_n$ .
  - If the *changeto* command is missing in  $b_n$  for  $\sigma_n$ , then the next tape  $T_{n+1} = T_n$ . In the complete module  $m_n$ , the case for  $\sigma_n$  will have the command *changeto*  $\sigma_n$ . So, the next tape is given by:

$$S_{n+1}(x) = \begin{cases} S_n(x) & x \neq j_n \\ \sigma_n & \text{otherwise} \end{cases}.$$

Therefore, we have  $S_{n+1} = S_n$  as well. So,  $T_{n+1} = S_{n+1}$ . Otherwise, we have the same *changeto* command in the two blocks, in which case  $T_{n+1} = S_{n+1}$  as well.

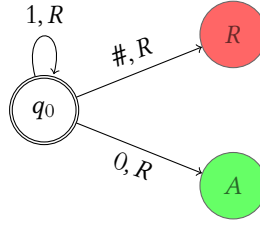
- If the *move* command is missing in  $b_n$  for  $\sigma_n$ , then the next tapehead index  $i_{n+1} = i_n - 1$ . In the complete module  $m_n$ , the case for  $\sigma_n$  will have the command *move left*, so we also have  $j_{n+1} = j_n - 1$ . Applying the inductive hypothesis, we have  $i_{n+1} = j_{n+1}$ . Otherwise, we have the same *move* command, meaning that  $i_{n+1} = j_{n+1}$  as well.
- We now consider the next block  $b_{n+1}$ :
  - \* If the block  $b_n$  is a *switch* block with a *while* case for  $\sigma_n$ , then this is still true in the module  $m_n$ . So, the next block to be executed in  $P$  is  $b_n$ , and the next module to be executed in  $P^+$  is  $m_n$ . In that case, the corresponding module of the block  $b_{n+1} = b_n$  is still  $m_{n+1} = m_n$ .
  - \* Instead, if the block  $b_n$  has no *flow* command for  $\sigma_n$ , and is not the last block, then the next block to execute is the block just below  $b_n$ , referred as  $b_{n+1}$ . By the definition of  $P^+$ , we find that the case block in the module  $m_n$  has a *goto* command, going to the module  $m_{n+1}$  which corresponds to the block  $b_{n+1}$ .
  - \* Now, if the *flow* command is missing for  $\sigma_n$  and this is the last block, then execution is terminated with the status *reject* for the program  $P$ . In that case, the case for  $\sigma_n$  in the module  $m_n$  has the *reject* command present, so the same happens for  $P^+$  as well.
  - \* Otherwise, both  $P$  and  $P^+$  have the same *flow* command, meaning that there is either correspondence between the next module to be executed, or both the program terminate with the same status.

In that case,  $P$  and  $P^+$  execute on  $T$  the same way by induction.  $\square$

Because of the equivalence between valid and complete programs, we will assume that every valid program is complete from now on.

## B.2 Equivalence of TMs and TMLs

In this section, we will show that there is an equivalence between TMs and valid TML programs. We will first construct a valid TML program for a TM and then show that it has the same



**Figure B.1:** A TM that accepts binary strings containing 0

behaviour as the TM. Later, we will construct a TM for a complete TML program, and show the equivalence in this case as well.

We will first illustrate how to convert a TM to a (complete) TML program. So, consider the TM at Figure B.1. Then, its corresponding TML program is the following:

```

1  alphabet = {0, 1}
2  module has0 {
3    if 1 {
4      changeto 1
5      move right
6      goto has0
7    } if 0 {
8      changeto 0
9      move right
10     accept
11   } if blank {
12     changeto blank
13     move left
14     reject
15   }
16 }

```

In general, we convert each (non-terminating) state in the TM  $M$  to a TML module. The following is how we create the module:

- the module contains a single *switch* command;
- for each letter  $\sigma$  in the alphabet  $\Sigma^+$ , denote  $\delta(q, \sigma) = (q', \sigma', \text{dir})$ . We add a case in the *switch* command corresponding to letter  $\sigma$  with the following commands:
  - *changeto*  $\sigma'$
  - *move* *dir*
  - if  $q'$  is *accept*, then the command *accept*; if  $q'$  is *reject*, then the command *reject*; otherwise, *goto*  $q'$ .

Moreover, we can construct the program  $P$  with:

- the alphabet  $\Sigma$ ;
- modules corresponding to every state  $q$  in  $M$ ;
- the module corresponding to the initial state  $q_0$  placed at the top.

We say that  $P$  is **the corresponding program for  $M$** .

**Theorem 2.** Let  $M$  be a TM, and let  $P$  be the corresponding program for  $M$ . Then,  $M$  and  $P$  execute on every tape  $T$  in the same way. That is,

- for every valid index  $n$ , if we have tape  $T_n$ , tapehead index  $i_n$  and module  $m_n$  for the TM program  $P$ , and we have tape  $S_n$ , tapehead index  $j_n$  and state  $q_n$  for the TM  $M$ , then  $T_n = S_n$ ,  $i_n = j_n$  and  $m_n$  is the corresponding module for  $q_n$ ;
- $M$  terminates execution on  $T$  if and only if  $P$  terminates execution on  $T$ , with the same final status (**accept** or **reject**).

*Proof.* We prove this by induction on the execution step.

- At the start, we have the same tape  $T$  for both  $M$  and  $P$ , with tapehead index 0. Moreover, the first module in  $P$  corresponds to the initial state  $q_0$ . So, the result is true if  $n = 0$ .
- Now, assume that the result is true for some integer  $n$ , where the TM state  $q_n$  is not **accept** or **reject**. In that case,  $T_n = S_n$ ,  $i_n = j_n$  and  $m_n$  is the corresponding module for  $q_n$ . Let  $\sigma_n$  be the letter at index  $i_n = j_n$  on the tape  $T_n = S_n$ . Denote  $q(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \text{dir})$ . In that case,

$$T_{n+1}(x) = \begin{cases} T_n(x) & x \neq i_n \\ \sigma_{n+1} & \text{otherwise,} \end{cases} \quad i_{n+1} = \begin{cases} i_n - 1 & \text{dir} = \text{left} \\ i_n + 1 & \text{dir} = \text{right,} \end{cases}$$

and the next state is  $q_{n+1}$ .

- We know that the module  $m_n$  in TM program  $P$  corresponds to the state  $q_n$ , so it has a **changeto**  $\sigma_{n+1}$  command for the case  $\sigma_n$ . In the case, the next tape for  $P$  is:

$$S_{n+1}(x) = \begin{cases} S_n(x) & x \neq i_n \\ \sigma_{n+1} & \text{otherwise.} \end{cases}$$

So,  $T_{n+1} = S_{n+1}$ .

- Similarly, the case also contains a **move dir** command. This implies that the next tapehead index for  $P$  is:

$$j_{n+1} = \begin{cases} j_n - 1 & \text{dir} = \text{left} \\ j_n + 1 & \text{dir} = \text{right.} \end{cases}$$

Hence,  $i_{n+1} = j_{n+1}$ .

- Next, we consider the value of  $q_{n+1}$ :
  - \* If  $q_{n+1} = q_n$ , then the case block is a *while* block, and vice versa. So, the next module to be executed is  $m_n$ . In that case,  $m_{n+1}$  still corresponds to  $q_{n+1}$ .
  - \* Otherwise, we have an *if* block.
    - In particular, if  $q_{n+1}$  is the **accept** state, then the case for  $\sigma_n$  contains the *flow* command **accept**, and vice versa. In that case, execution terminates with the same final status of **accept**. The same is true for **reject**.
    - Otherwise, the module contains the command **goto**  $m_{n+1}$ , where  $m_{n+1}$  is the corresponding module for  $q_{n+1}$ .

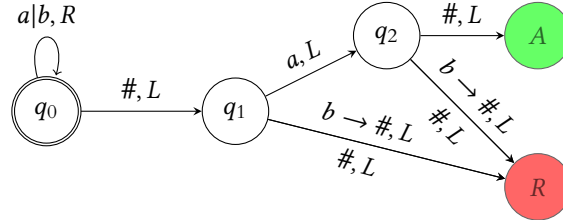
In that case,  $P$  and  $M$  execute on  $T$  the same way by induction.  $\square$

Next, we construct a TM for a (complete) TML program. This process is essentially the inverse of the one we saw converting a complete TML program to a TM. In particular, for each module  $m$  in  $P$ , we construct the state  $q$  as follows- for each letter  $\sigma$  in  $\Sigma^+$ , we define  $\delta(q, \sigma) = (q', \sigma', \text{dir})$ , where:

- the value  $\sigma'$  is the letter given in the *changeto* command within  $m$ ;
- the value **dir** is the direction given in the *move* command within  $m$ ;

- if the *flow* command in  $m$  corresponding to  $\sigma$  is **accept**, then  $q'$  is the **accept** state; if it is **reject**, then  $q'$  is the **reject** state; if we are in a *while* block, then  $q' = q$ ; otherwise,  $q'$  is the state corresponding to the module given in the *goto* command.

Then, the TM with all the states  $q$ , the same alphabet  $\Sigma$ , the transition function  $\delta$  and initial state  $q_0$  corresponding to the first module in  $P$  is the **corresponding TM for  $P$** .



**Figure B.2:** The TM corresponding to the program above. The state  $q_0$  corresponds to the module *moveToEnd*; the state  $q_1$  corresponds to the module *checkAFirst*; and the state  $q_2$  corresponds to the module *checkASecond*.

We now illustrate this process with an example. So, consider the following complete TM program:

```

1  alphabet = {a, b}
2  module moveToEnd {
3    while a {
4      changeto a
5      move right
6    } while b {
7      changeto b
8      move right
9    } if blank {
10     changeto blank
11     move left
12     goto checkAFirst
13   }
14 }
15 module checkAFirst {
16   if a {
17     changeto blank
18     move left
19     goto checkASecond
20   } if b, blank {
21     changeto blank
22     move left
23     reject
24   }
25 }
26 module checkASecond {
27   if a {
28     changeto blank
29     move left
30     accept
31   } if b, blank {
32     changeto blank
33     move left
34     reject
35   }

```

Then, its corresponding TM is given in Figure B.2.

**Theorem 3.** *Let  $P$  be a complete TM program, and let  $M$  be the corresponding TM for  $P$ . Then,  $P$  and  $M$  execute on every tape  $T$  in the same way. That is,*

- *for every valid index  $n$ , if we have tape  $T_n$ , tapehead index  $i_n$  and module  $m_n$  for TM program  $P$ , and we have tape  $S_n$ , tapehead index  $j_n$  and state  $q_n$  for the TM  $M$ , then  $T_n = S_n$ ,  $i_n = j_n$  and  $q_n$  is the corresponding state for  $m_n$ ;*
- *$P$  terminates execution on  $T$  if and only if  $M$  terminates execution on  $T$ , with the same final status (accept or reject).*

*Proof.* We prove this as well by induction on the execution step of the tape.

- At the start, we have the same tape  $T$  for both  $P$  and  $M$ , with tapehead index 0. Moreover, the initial state  $q_0$  in  $M$  corresponds to the first module in  $P$ . So, the result is true if  $n = 0$ .
- Now, assume that the result is true for some integer  $n$ , which is not the terminating step in execution. In that case,  $S_n = T_n$ ,  $j_n = i_n$  and  $q_n$  is the corresponding state for  $m_n$ . Let  $\sigma_n$  be the letter at index  $j_n = i_n$  on the tape  $S_n = T_n$ . We now consider the single switch block in  $m_n$ :

- If the block in  $m_n$  corresponding to  $\sigma_n$  is a *while* block, then we know that its body is partially complete, and so is composed of the following commands:

\* **changeto**  $\sigma_{n+1}$   
 \* **move dir**

So, we have  $\delta(q_n, \sigma_n) = (q_n, \sigma_{n+1}, \text{dir})$ . Using the same argument as in Theorem 2, we find that  $T_{n+1} = S_{n+1}$  and  $i_{n+1} = j_{n+1}$ . Also,  $q_{n+1} = q_n$  is the corresponding state for  $m_{n+1} = m_n$ .

- Otherwise, we have an *if* command. In this case, the case body is complete, and so composed of the following commands:

\* **changeto**  $\sigma_{n+1}$   
 \* **move dir**  
 \* **accept, reject or goto**  $m_{n+1}$ .

So, we have  $\delta(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \text{dir})$ , where  $q_{n+1}$  is the corresponding state to the *flow* command present. Here too, we have  $T_{n+1} = S_{n+1}$  and  $i_{n+1} = j_{n+1}$  by construction.

Now, we consider the flow command:

- If we have an **accept** command in the body, then  $q_{n+1}$  is the accepting state, and vice versa. So, we terminate execution with the final status of **accept**. The same is true for **reject**.
- Otherwise, the state  $q_{n+1}$  is the corresponding state to the module  $m_{n+1}$ .

In all cases, there is a correspondence between the state for  $m_{n+1}$  and  $q_{n+1}$ .

So, the result follows from induction. □

Hence, we have established that for any valid TML program, there is a TM, and vice versa.

### B.3 TML as a model of computation

Since TML programs and TMs are equivalent, this implies that TML programs are a model for computation. Note that while we have given a proof for equivalence for TMs, this representation is based on accepting and rejecting programs only. Not all TMs are of this form. In particular, it is also possible for a TM to halt instead of accepting or rejecting.

Although the equivalence is limited to a subclass of TMs, we can add another flow command `halt` that mimics the halting behaviour. However, this is not necessary- we can use the accept state (or equally the reject state) to mimic the behaviour of halting. Since accepting or rejecting results in the program halting, we can simply disregard the final result, and possibly read the output from the tape to infer the actual result.



## C | Product Screenshots

In this chapter, we illustrate some screenshots of the website.

### C.1 Homepage

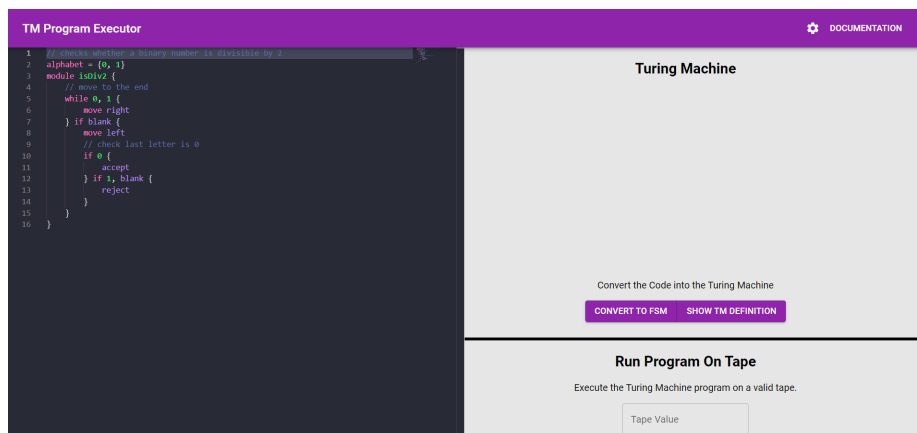


Figure C.1: The initial rendering of the homepage.

The initial rendering of the homepage is given in Figure C.1. It shows an example program initially, with no TM conversion or tape execution. We can change the program in settings. The user can press the button in TM panel to convert the program to either the FSM representation or the definition version.

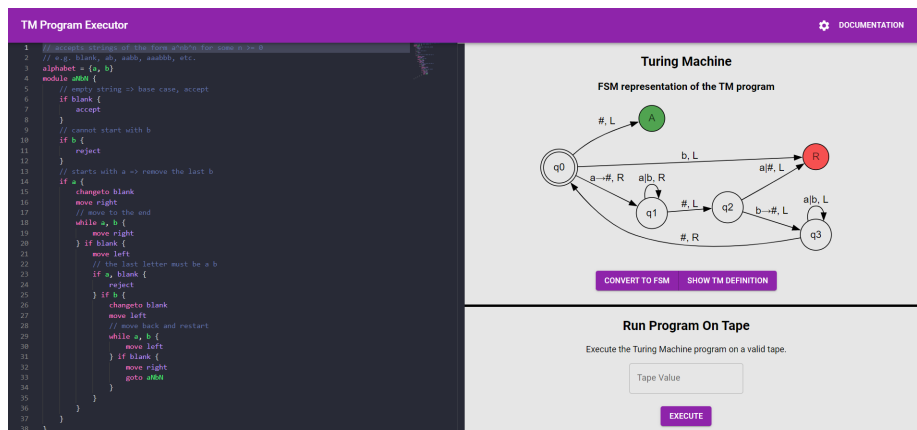


Figure C.2: The homepage after the user converts the TML program into FSM version of TM.

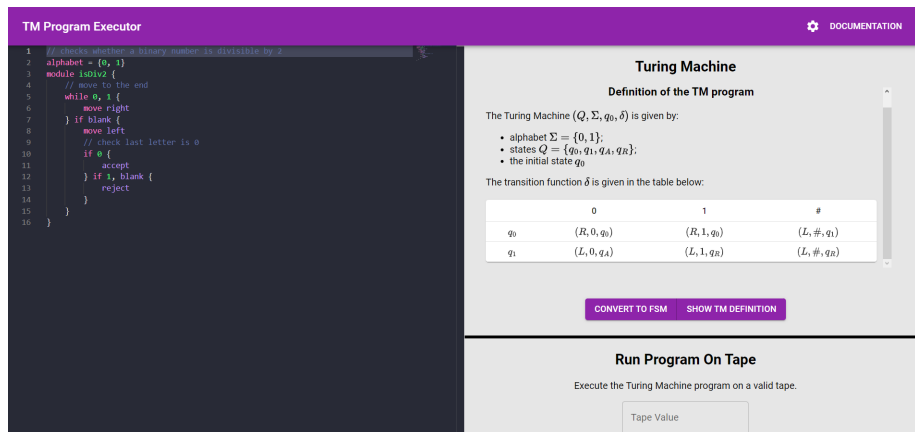


Figure C.3: The homepage after the user converts the TML program into definition version of TM

Figure C.2 shows the FSM conversion of a TML program on the page, while figure C.3 shows the definition version.

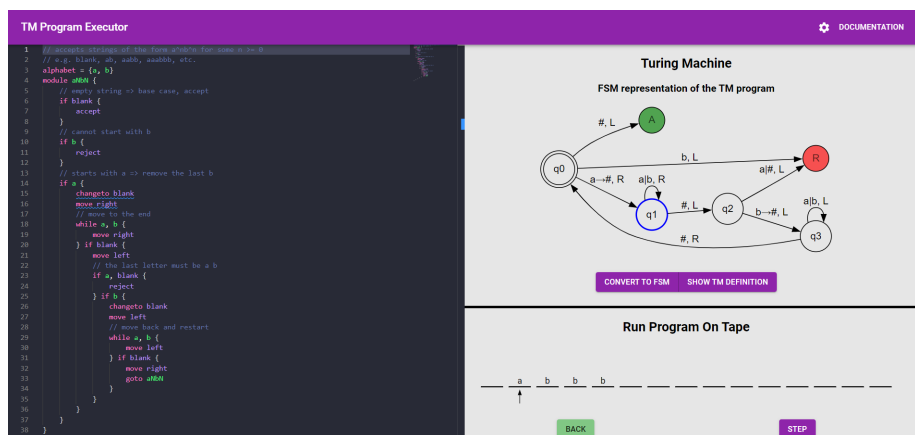


Figure C.4: The homepage during tape execution.

Figure C.4 shows the website during tape execution. The execution is illustrated in code- the current block is highlighted. Moreover, since TM has been converted, the current state is also highlighted in blue.

## C.2 Documentation Pages

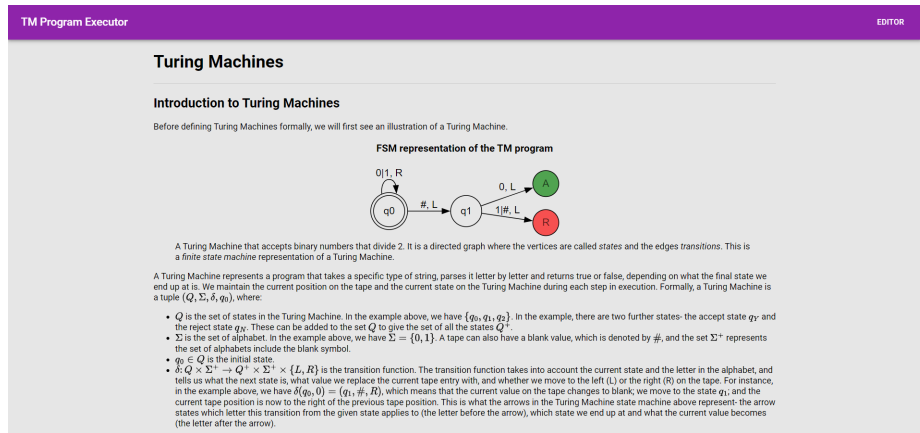


Figure C.5: The documentation webpage for TM

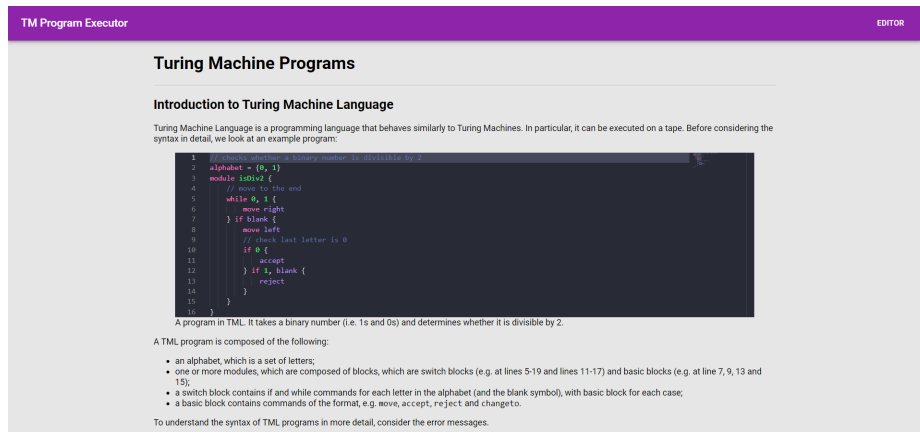


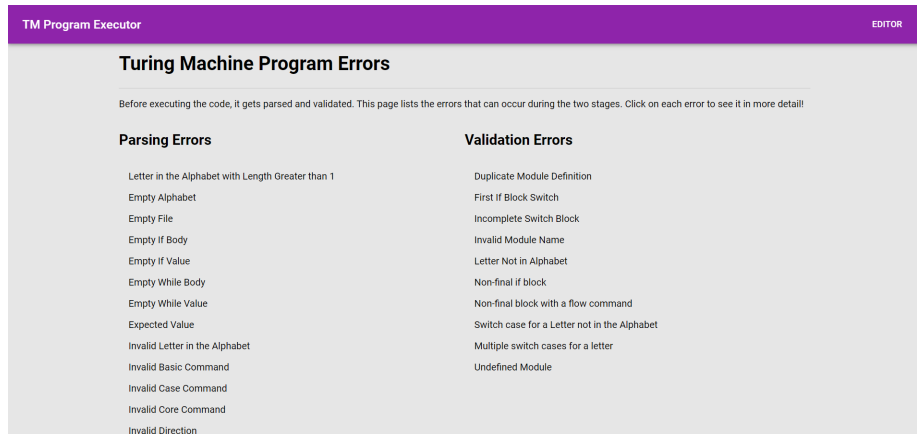
Figure C.6: The documentation webpage for TML

There are 2 pages in the site dedicated to documentation- one that documents TMs and the other that documents TML. The initial definitions are shown in Figures C.5 and C.6. In each page, there is an example before the actual definition.

Below the definition, we describe execution on a tape. The user can then execute the TM/TML program by inputting a value. Like in the homepage, different aspects are highlighted during execution to make it easier to follow.

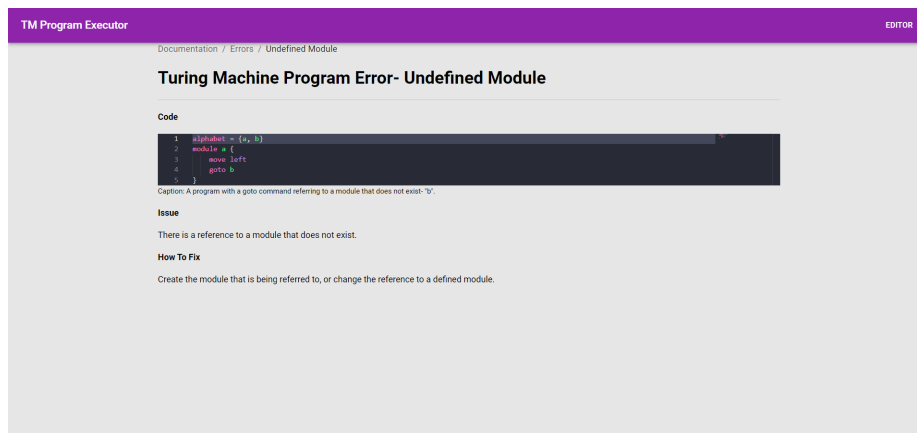
## C.3 Program Error Pages

There is a specific page in the website that lists all the errors– this is shown in Figure C.7. The errors are partitioned into 2 groups– parsing/syntax errors and validation errors.



*Figure C.7: The general error page that lists all the syntax and validation errors.*

Clicking on an error takes the user to the specific error page. The error page for an undefined module is given in Figure C.8. At the top of the webpage is a program that also has this error. We describe the error and then explain how to resolve it.



*Figure C.8: The error page for an undefined module.*

## D | Evaluation Content

### D.1 Worksheet

#### D.1.1 Introduction to Turing Machine Language

In this section, you are given some programs in Turing Machine Language (TML). They will be used to explain the syntax of the programming language and how they can be run on tapes.

- isDiv2:

```

1 // checks whether a binary number is divisible by 2
2 alphabet = {0, 1}
3 module isDiv2 {
4     // move to the end
5     while 0, 1 {
6         move right
7     } if blank {
8         move left
9         // check last letter is 0
10        if 0 {
11            accept
12        } if 1, blank {
13            reject
14        }
15    }
16 }
```

- isDiv2Rec:

```

1 // checks whether a binary number is divisible by 2, recursively
2 alphabet = {0, 1}
3 module isDiv2Rec {
4     // recursive case: not at the end => move closer to the end
5     if 0, 1 {
6         move right
7         goto isDiv2Rec
8     }
9     // base case: at the end => check final letter 0
10    if blank {
11        move left
12        if 0 {
13            accept
14        } if 1, blank {
15            reject
16        }
17    }
```

```
18 }
```

Both `isDiv2` and `isDiv2Rec` correspond to the TM in Figure D.1.

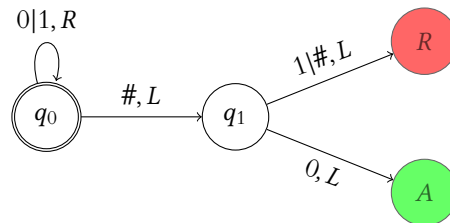


Figure D.1: The TM for `isDiv2` and `isDiv2Rec`.

• `aNbN`:

```

1 // accepts strings of the form a^nb^n for some n >= 0
2 // e.g. blank, ab, aabb, aaabbb, etc.
3 alphabet = {a, b}
4 module aNbN {
5     // empty string => base case, accept
6     if blank {
7         accept
8     }
9     // cannot start with b
10    if b {
11        reject
12    }
13    // starts with a => remove the last b
14    if a {
15        changeto blank
16        move right
17        // move to the end
18        while a, b {
19            move right
20        } if blank {
21            move left
22            // the last letter must be a b
23            if a, blank {
24                reject
25            } if b {
26                changeto blank
27                move left
28                // move back and restart
29                while a, b {
30                    move left
31                } if blank {
32                    move right
33                    goto aNbN
34                }
35            }
36        }
37    }
38 }
```

The program  $aNbN$  corresponds to the TM in Figure D.2.

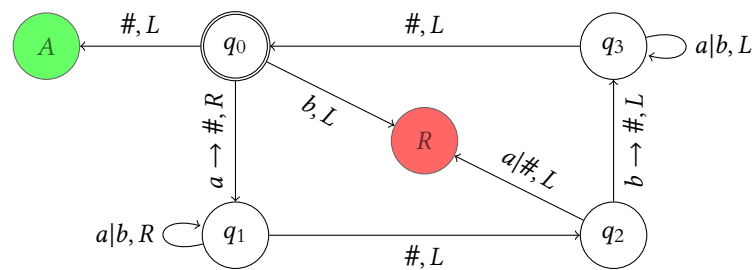


Figure D.2: The TM for  $aNbN$ .

### D.1.2 Identifying TML Programs

In this section, you are presented with TML programs. You will be given some tape values to run the program in and decode what values the program accepts. You are encouraged to use the website to try and solve this.

1. Consider the following TML Program:

```

1  alphabet = {0, 1}
2  module mystery1 {
3    while 0, 1 {
4      move right
5    } if blank {
6      move left
7      if blank, 0 {
8        reject
9      } if 1 {
10     move left
11     if blank, 1 {
12       reject
13     } if 0 {
14       accept
15     }
16   }
17 }
18 }

```

- Does the program accept the values:
  - 10 (NOTE: This is 2 in decimal)
  - 1
  - 100 (NOTE: This is 4 in decimal)
  - 101 (NOTE: This is 5 in decimal)
  - 110 (NOTE: This is 6 in decimal)
- Describe the values this program accepts.

2. Consider the following TML program:

```

1  alphabet = {a, b}
2  module mystery2 {
3      if blank {
4          accept
5      } if b {
6          reject
7      } if a {
8          changeto blank
9          move right
10         if b, blank {
11             reject
12         } if a {
13             changeto blank
14             move right
15             while a, b {
16                 move right
17             } if blank {
18                 move left
19                 if a, blank {
20                     reject
21                 } if b {
22                     changeto blank
23                     move left
24                     if a, blank {
25                         reject
26                     } if b {
27                         changeto blank
28                         move left
29                         while a, b {
30                             move left
31                         } if blank {
32                             move right
33                             goto mystery2
34                         }
35                     }
36                 }
37             }
38         }
39     }
40 }

```

- (a) Does the program accept the values:
- i. *ab*
  - ii. *aabb*
  - iii. *abba*
  - iv. *bab*
- (b) Describe the values this program accepts.



### D.1.3 Identifying TMs

In this section, you are presented with TMs. You will be given some tape values to run the program in and decode what values the program accepts. Since the website can only execute TML programs, you are also given the TML program for the code, but it is not comprehensible like the previous programs; you will likely find it easier to understand the TM than the program (which you should do!).

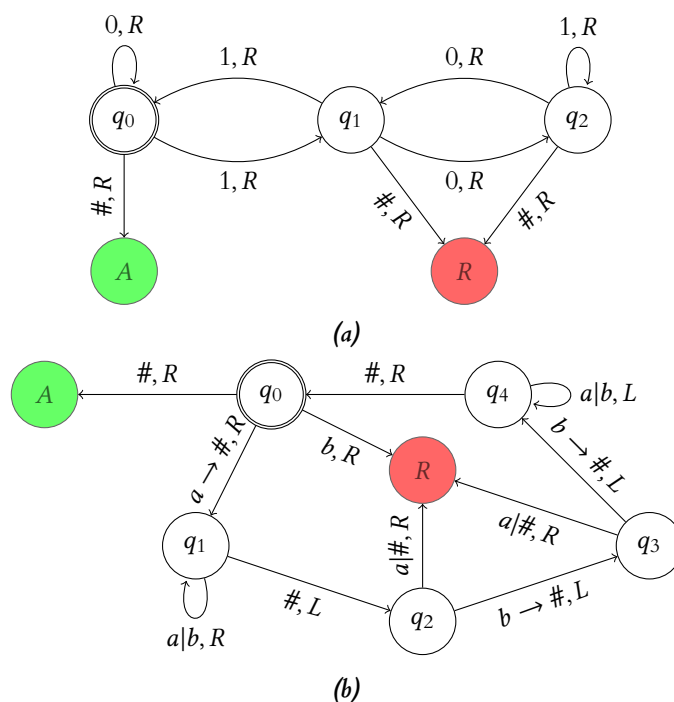


Figure D.3: 2 Turing Machines

1. Consider the TM FSM at Figure D.3 (a). You are given a basic representation of this TM as code in Teams. The file is called mystery3.
  - (a) Does the TM accept the values:
    - i. 11 (NOTE: This is 3 in decimal)
    - ii. 10 (NOTE: This is 2 in decimal)
    - iii. 1
    - iv. 110 (NOTE: This is 6 in decimal)
    - v. 1001 (NOTE: This is 9 in decimal)
  - (b) Describe the values this program accepts.
2. Consider the TM FSM at Figure D.3 (b). You are given a basic representation of this TM as code in Teams. The file is called mystery4.
  - (a) Does this TM accept the values:
    - i. *ab*
    - ii. *abb*
    - iii. *aabbbb*
    - iv. *bab*
    - v. *abba*
  - (b) Describe the values this program accepts.

### D.1.4 Writing TML Programs

Following a similar syntax to the code given above, write the following programs. You are free to use the website to check the accuracy of the program while writing the programs. Please answer these questions in the survey.

1. divisibility by 4 in binary iteratively [HINT: Go to the end and check for 2 zeros. Allow 0 as well.]
2. divisibility by 4 in binary, recursively.

The remaining questions are optional.

3. strings of the form  $a^n b^m c^{n+m}$
4. strings of the form  $a^n b^n c^n$
5. HARD: check there are same number of  $a$ 's and  $b$ 's

## D.2 Checklist

## D.3 Survey

**School of Computing Science  
University of Glasgow**

**Ethics checklist form for 3<sup>rd</sup>/4<sup>th</sup>/5<sup>th</sup> year, and taught MSc projects**

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

**If no other people have been involved in the collection of information, then you do not need to complete this form.**

If your evaluation does not comply with any one or more of the points below, please contact the Chair of the School of Computing Science Ethics Committee ([matthew.chalmers@glasgow.ac.uk](mailto:matthew.chalmers@glasgow.ac.uk)) for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

- 
1. Participants were not exposed to any risks greater than those encountered in their normal working life.

*Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback*

2. The experimental materials were paper-based, or comprised software running on standard hardware.  
*Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.*

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.

*If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.*

*Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.*

4. No incentives were offered to the participants.

*The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.*

5. No information about the evaluation or materials was intentionally withheld from the participants.  
*Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.*
6. No participant was under the age of 16.  
*Parental consent is required for participants under the age of 16.*
7. No participant has an impairment that may limit their understanding or communication.  
*Additional consent is required for participants with impairments.*
8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.  
*A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.*
9. All participants were informed that they could withdraw at any time.  
*All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.*
10. All participants have been informed of my contact details.  
*All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.*
11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.  
*The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.*
12. All the data collected from the participants is stored in an anonymous form.  
*All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.*

---

Project title Turing Machine Language

Student's Name Pete Grawam

Student Number 24814716

Student's Signature Pete

Supervisor's Signature Oruela Dordie

Date 8/2/2023

# Pete Evaluation Survey

This is the follow-up survey to the worksheet. There are 2 parts to this- evaluating the programming language and the website.

\* Required

1. How many years of programming experience do you have? \*

- ☐ 0-1 Year
- ☐ 1-2 Years
- ☐ More than 3 Years

2. Were you familiar with Turing Machines before the workshop? \*

- ☐ Familiar
- ☐ Somewhat familiar
- ☐ Not familiar

## Final Questions

3. Iterative program that checks for divisibility by 4

4. Recursive program that checks for divisibility by 4

5. Strings of the form  $a^n b^m c^{n+m}$

6. Strings of the form  $a^n b^n c^n$

7. Check there are the same number of a's and b's

## The Programming Language

8. Evaluate the Turing Machine Language (TML) under the following categories. Note that the focus here is on the **language**, and not the website. \*

	Strongly Disagree	Disagree	Agree	Strongly Agree
TML is easy to understand	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
TML is easy to write programs in	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I was able to fix errors in my code using the error messages provided	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I was able to easily reason executing a program on a tape	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

9. Do you have any general comments about the language?



10. Are there any features that should be added to the language in your opinion, e.g. moving to the end of the tape in one line of code?

11. Compare programs in Turing Machine language (TML) with Turing Machines (TM) \*

	Strongly Disagree	Disagree	Agree	Strongly agree
I am more confident in writing a program in TML than drawing a TM	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I find it easier to reason what a TML program accepts than a TM	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

12. If you had to draw a TM, would you write a TML program for it first? \*

- ☐ Yes
- ☐ No
- ☐ Maybe

## Website Evaluation

13. Evaluate the website under the following categories: \*

	Strongly disagree	Disagree	Agree	Strongly agree
The website is easy to follow	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The presentation of the website is intuitive	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
There were no visible bugs in the website	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The website was fast	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The website feels complete	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The code execution was easy to follow	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The code editor was easy to use	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

14. The website presents tape execution using pop-ups. Do you think this is a good idea or should it be replaced by just a short paragraph within the tape section, or anything else? \*

- ☐ Pop-ups are fine!
- ☐ A short paragraph is a better idea
- ☐ Other

15. Are there any features you would like to be added to the website?

---

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

# Bibliography

Aho, A. V., Sethi, R. and Ullman, J. D. (2007), *Compilers: principles, techniques, and tools*, Vol. 2, Addison-wesley Reading.

Copeland, B. J. (2004), *The essential Turing*, Clarendon Press.

Gamma, E., Johnson, R., Helm, R., Johnson, R. E. and Vlissides, J. (1995), *Design patterns: elements of reusable object-oriented software*, Pearson Deutschland GmbH.

Hopcroft, J. E., Motwani, R. and Ullman, J. D. (2001), *Introduction to automata theory languages and computation*, Addison-Wesley, United States of America.

Turing, A. M. (1936), 'On computable numbers, with an application to the Entscheidungsproblem', *J. of Math* **58**(345–363), 5.