University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Turing Machine Language

Pete Gautam

March 18, 2023

Abstract

Turing Machines are taught to students who have some experience in programming. This is typically done in a diagrammatic or a mathematical manner. This projects defines a programming language for Turing Machines and investigates if it is easier for them to grasp the concepts using programming or diagrams. It was found that the language was somewhat easier for students to learn than the diagrammatic approach.

Acknowledgements

I would like to thank my supervisor Ornela for all the support she provided throughout the semester. Also, I am grateful for everyone who participated in the evaluation sessions- committing one hour in person was a big thing to ask and I am glad so many came along!

Education Use Consent

Contents

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

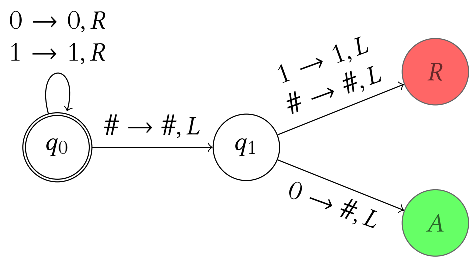
2.1 Turing Machine

2.1.1 Introduction to Turing Machines

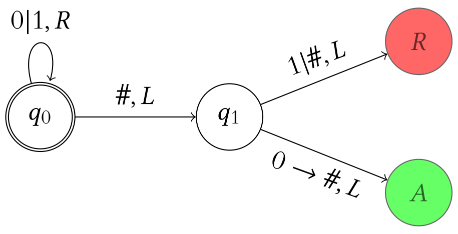
A **Turing Machine** (TM) is a collection (Q, Σ, δ, q_0), where:

- Q is a set of **states**, including the **accept state** A and **reject state** R;
- Σ is the set of **letters**, which does not include the `blank` symbol;
- $\delta \colon Q \setminus \{A, R\} \times \Sigma^+ \to Q \times \Sigma^+ \times \{\texttt{left}, \texttt{right}\}$, where $\Sigma^+ = \Sigma \cup \{\texttt{blank}\}$, 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).



(a) Full notation



(b) Shorthand notation

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 δ, e.g. $\delta(q_1, 0) = (A, \texttt{blank}, \texttt{left})$.

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

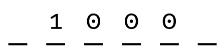


Figure 2.2: A TM tape on $\{0, 1\}$.

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

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

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 Σ, and let T be a tape on Σ. We execute M on T inductively, as follows:

- At any point during execution, we maintain 3 objects: a tape on Σ; 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 $\delta(q, t) = (q', t', \texttt{dir})$. 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

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

  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'.

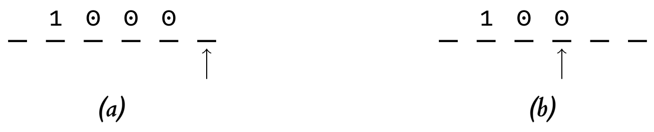


Figure 2.3: Some of the tape states during execution.

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 $\delta(q_1, 0) = (A, \texttt{blank}, L)$. 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 λ-calculus are equivalent. Hence, it follows that λ-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)`.

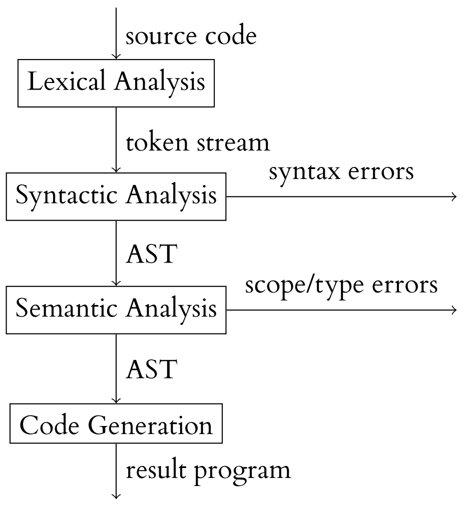


Figure 2.4: *The data flow between the compilation phases.*

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.

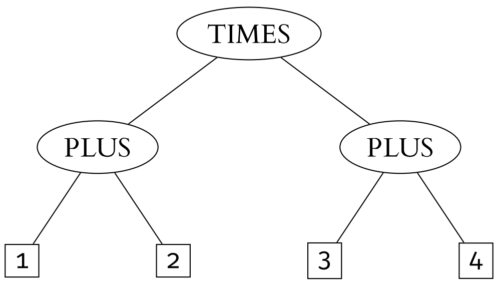


Figure 2.5: *The AST for the expression* `(1 + 2) * (3 + 4)`

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 λ-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.

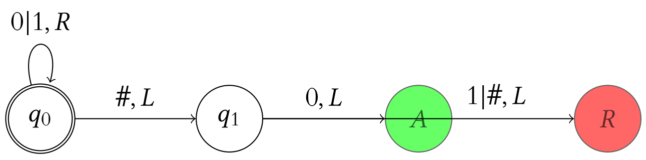


Figure 3.1: A possible initial rendering of a FSM

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 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 simulates a state in TMs, although it is more expressible than a state. We want modules to also be thought of as functions in a traditional PL. To allow for flow of code to go from one module to another, we 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 }
```

The execution of a program starts at 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` commands. This is shown in the example below.

```
if a {
   move right
   accept
} if b, blank {
   changeto blank
}
```

Although the language is already equivalent to TMs, the programs that we can currently write 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 12-15.

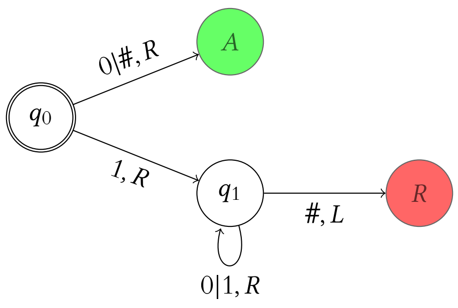


Figure 4.1: *A TM with a self-loop at the state q_1*

Although nesting has made the language more like a typical PL, this is not enough. In particular, if we have a self-loop at a non-starting state, then the program cannot be written compactly. 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:

```
alphabet = {0, 1}
module q0 {
   if 0, blank {
      move right
      accept
   } if 1 {
      move right
      goto q1
   }
}
```

```
module q1 {
   if 0, 1 {
      move right
      goto q1
   } if blank {
      move left
      reject
   }
}
```

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- because the block corresponding to q_1 would be nested within q_0, recursion would convert the self-loop at q_1 into a transition from q_1 to q_0. We want there to be another way to represent this program that resembles a traditional PL better.

To allow for self-loops to be nested, we introduce a new construct- a `while` command. This is similar to an `if` command, 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:

```
alphabet = {0, 1}
module program {
   if 0, blank {
      move right
      accept
   } if 1 {
      move right
      while 0, 1 {
         move right
      } if blank {
         move left
         reject
      }
   }
}
```

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 tape 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 a TML program, and a 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

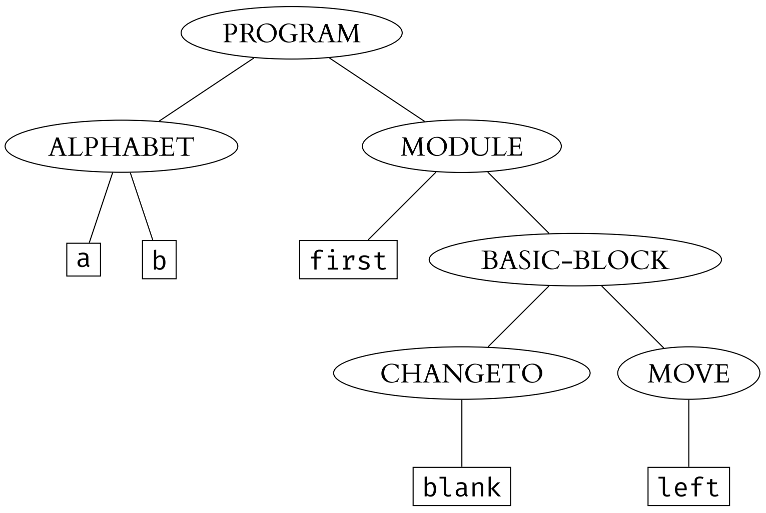


Figure 4.2: An AST for the TML program with a module called `first`.

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. So, assume we have the following source code.

```
alphabet={a, b}
module first {
   changeto blank
   move left
}
```

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

The parser is top-down 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. Because the language does not have any complex parsing rules, we also make use of predictive parsing.

We construct the AST given as follows:

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 a succinct message.

4.2.3 Semantic Analysis

After the AST has been constructed, we perform semantic analysis. The TML does not have a type system, meaning that we do not 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 check that a module is not defined twice, and is not called `accept` or `reject`. We also validate that the letter of a `changeto` command is one of the letters in the `alphabet` or `blank`.

Moreover, there is also a check to ensure that a *switch* block contains precisely one case for each letter in the alphabet. That is, there are no duplicate cases present, 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.

This process is different to the one described in the appendix- here, we are directly converting a valid TML program into a TM. In essence, we have combined the two steps given in the appendix to achieve this.

Initially, we define a TM. During the traversal of the AST, we fill it with relevant states and transitions. This mostly takes place when we are at a block, either inside a *module* or an *if* command. The process depends on the type of block we have:

- if we have a *switch* block, then we define the transition function one by one for each letter, by visiting all the cases within the block;
- if we have a *basic* block, then we define the transition function for all the letters in one go.

We have commands within the block/case that we can use to define the transition. For instance, if we have the command `move left`, then the transition function will report move to the left. If the command is not present, then we add the default transition, as specified in the language specification given in the appendix.

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.

The execution of a TML program follows the rules given in the specification. This is included in the appendix. Similarly, the execution of a TM follows the rules given in the background section.

4.3 Product

4.3.1 Structure

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

- the **homepage** that allows the user to make use of the parser;
- the **documentation** pages that explains TMs and TML programs; along with
- the **error** pages to illustrate syntax and validation (non-syntax) errors.

A screenshot of the homepage is given in Figure 4.3. More screenshots are in the appendix.

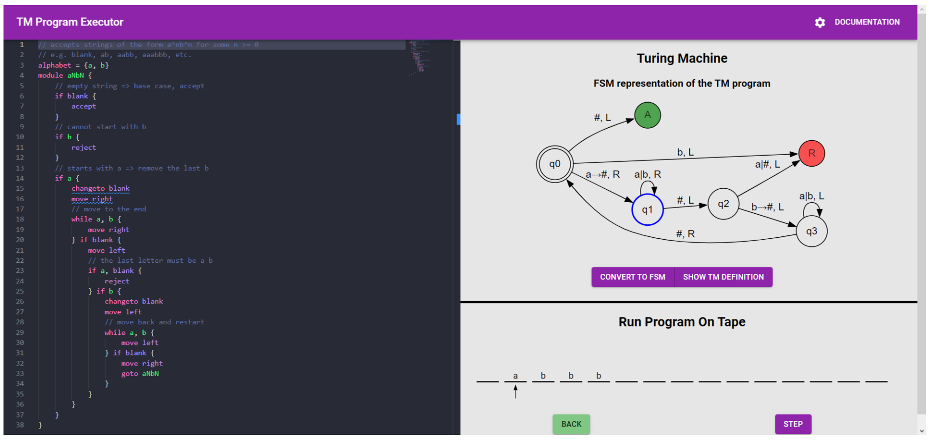

```
// accepts strings of the form a^nb^n for some n >= 0
// e.g. blank, ab, aabb, aaabbb, etc.
alphabet = {a, b}
module aNbN {
    // empty string => base case, accept
    if blank {
        accept
    }
    // cannot start with b
    if b {
        reject
    }
    // starts with a => remove the last b
    if a {
        changeto blank
        move right
        // move to the end
        while a, b {
            move right
        } if blank {
            move left
            // the last letter must be a b
            if a, blank {
                reject
            } if b {
                changeto blank
                move left
                // move back and restart
                while a, b {
                    move left
                } if blank {
                    move right
                    goto aNbN
                }
            }
        }
    }
}
```

Figure 4.3: The website homepage.

Homepage The user can input a program to the editor. If the program is valid, it can be compiled to a TM in two ways- it can either be converted to a FSM or to the definition version. The user can also execute the program on a tape after inputting a value. The **step** button performs one step in execution.

The toolbar features a button to go to the documentation pages. It also features a button that allows the user to configure the page, e.g. fill the editor with some example code, change the editor theme, change font size, etc.

Documentation Pages The website has documentation pages that define both TMs and TML programs. In particular, the website gives the formal definition, shows an example and allows the user to execute the example on a tape.

Error Pages For every language error, there is a dedicated page that:

- describes the error informally;
- illustrates the error with an example program; and
- presents a way to resolve the error.

There is also a general error page that lists all these errors.

The website makes use of **Material Design**. The Material design provides common-purpose components, such as a toolbar. Moreover, Material design is quite widespread since all Google products make use of it (Google (2023)). 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 design consistent.

4.3.2 TM Conversion

The website supports live conversion of a TML program into a TM. The user can convert the TML program into both the FSM representation of a TM and its definition.

The formal definition of the TM 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

	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.*

$\Sigma = \{0, 1\}$. 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 $\mathbf{q_0}$

4.3.3 Tape Execution

The user can input a valid string in the alphabet and execute the TML program on a tape. A valid string consists of letters in the alphabet, including the blank symbol. 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.

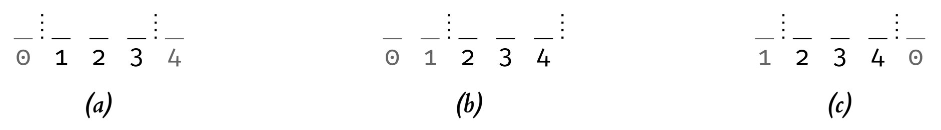


Figure 4.4: *The transition process for tape entries.*

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 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, the parser was expected to become an node package manager (NPM) package. Moreover, TML is quite a simple language, which made this task relatively easy and short.

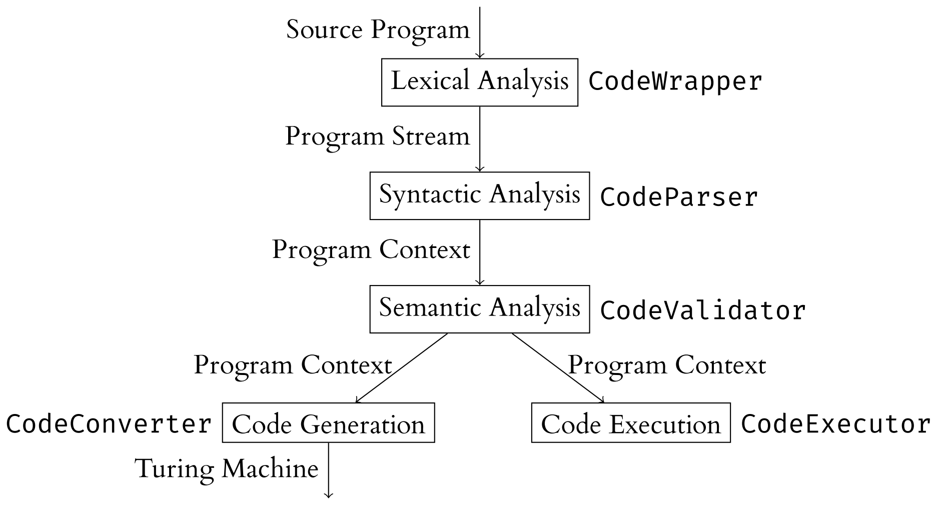


Figure 5.1: The parsing process. The process is given inside the box. The class used to achieve the process is given in label, outside of the box. The flow of data is also shown.

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`.

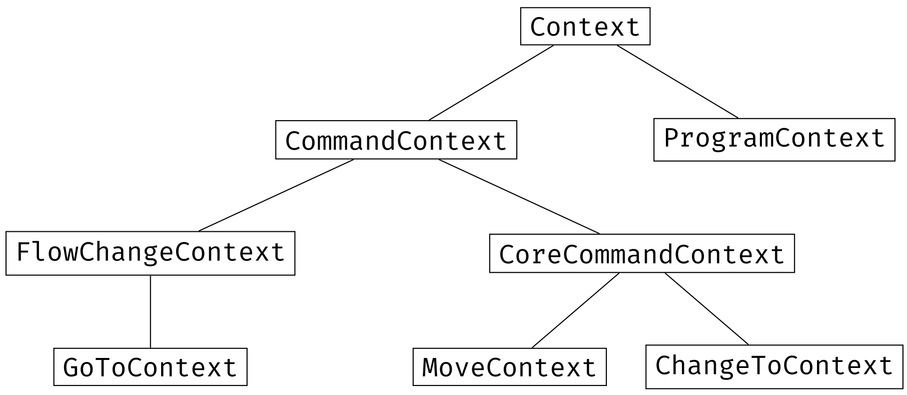


Figure 5.2: *A snippet of the `Context` class hierarchy. All the non-leaf classes are abstract.*

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, meaning that we delegate the parsing procedure to a method that parses a specific construct. For example, `parseModule` parses a *module*, while `parseIf` parses an *if* command. A snippet of the method `parseProgram` is given below.

```
parseProgram():ProgramContext {
   var alphabet:AlphabetContext = parseAlphabet();

   var modules:ModuleContext[] = [];
   while (this.code.moveNext()) {
      modules.add(parseModule());
   }

   return new ProgramContext(position, alphabet, modules);
}
```

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

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 this case, we want to create a method to validate each `Context`. Moreover, the semantic analysis is conducted by the class `CodeValidator`, which implements `CodeVisitor` to accommodate the visitor design pattern.

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

```
class GoToContext extends FlowChangeContext {
   identifier:string;

   visit<T>(visitor:CodeVisitor<T>):T {
      return visitor.visitGoTo(this);
   }

   // ... other methods for goto context
}
```

Each `Context` class makes use of the right visitor method given in `CodeVisitor`. Now, in the `CodeValidator` class, we can define the following method that 'visits' a *goto* statement and validates it.

```
class CodeValidator extends CodeVisitor<boolean> {
   visit(context:Context):boolean {
      return context.visit(this);
   }

   // validate that the goto identifier a module name
   visitGoTo(gotoContext:GoToContext):boolean {
      if (!moduleNames.contains(gotoContext.identifier)) {
         throw new CodeError("Undefined name- " + identifier);
      }

      return true;
   }

   // ... other visitor methods
}
```

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

When traversing the AST, we typically want to aggregate the result or share it with the parent. For this reason, we typically return a specific type of values within each visit method. To allow any type to be returned, the class `CodeVisitor` has a type parameter that represents the return type of each `visit` method.

In `CodeValidator`, we return `boolean` values. In particular, 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. In the example code above, a block with a *goto* command returns `true` since it is a *flow* command.

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. In particular, a TM is composed of many instances of `TMState`, which represent states within a TM. A TM state instance has a method `transition` that takes in a letter and returns the transition data, as a `TMChange`. A `TMChange` shows:

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

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. This is defined in a very complex manner, depending on whether we have an *if* or a *while* command, or none at all! Hence, we delegate this responsibility from a *switch* block to the relevant case command.

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 been implemented in the class `TMExecutor`. This also makes use of the iterator design pattern and supports execution one step at a time. The definition of TM execution is given in the background section.

5.2 Product

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

5.2.1 Website Framework

Initially, 3 frameworks were considered to implement the website:

- the **Webpack** framework - it has little overhead, allows for a lot of flexibility, but does not directly provide components (such as toolbar and drawer) or support state management (Webpack (2023));
- the **React** framework- it has significantly more overhead than Webpack, but it also provides a rich collection of components and supports state management directly (ReactJS (2023));
- the **Angular** framework- it has even more overhead than React, but, like React, it has a rich collection of components and supports state management (AngularJS (2023)).

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. Managing state manually would increase the complexity of the project, and could easily be avoided by choosing the right 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, including some of the APIs used in this project. This is true since React is more widely used than Angular (w3techs (2023)). The React framework supports coding in either JavaScript or TypeScript. The project made use of the TypeScript version for type safety.

5.2.2 Editor

The editor feature was implemented using the **monaco** API. The API was chosen since it integrates well with React and provides many features (Microsoft (2023)). The features include:

- syntax highlighting with different priorities (e.g. an error gets a high priority while code execution 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 within 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 (Stack Exchange (2022)).

5.2.3 FSM Generation

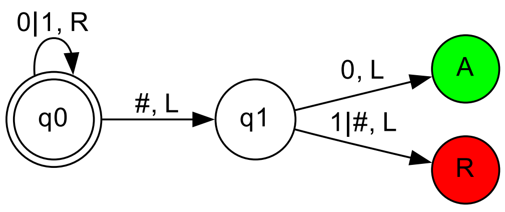


Figure 5.3: The graphviz rendering of the directed graph `isDiv2`.

The FSM representation of the TM is created using the **graphviz** framework. The API makes use of the **DOT** notation (Graphviz (2023*b*)). The DOT notation represents graphs in text by listing their nodes and edges (Graphviz (2023*a*)). A graph in DOT notation is given below:

```
digraph isDiv2 {
   node [shape = "doublecircle"]; q0;
   node [shape = "circle"]; q1;
   node [style = "filled", fillcolor = "green"]; A;
   node [style = "filled", fillcolor = "red"]; R;
   q0 -> q0 [label = "0|1, R"];
   q0 -> q1 [label = "#, L"];
   q1 -> A [label = "0, L"];
   q1 -> R [label = "1|#, L"];
}
```

This graph represents the FSM representation of a TM. In this graph:

- the keyword `digraph` implies that the graph is directed;
- the keyword `node` creates a state; and
- the arrow symbol (`->`) constructs edges between 2 states.

Nodes and edges take optional parameters within square brackets that allows them to be customised.

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 them with the corresponding transition (like in lines 6–9).

The advantage of using the graphviz package is that it can produce a well-formatted FSM. Initially, the graph was rendered by placing the states in order of its label, similar to Figure 3.1. 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!

5.2.4 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`. The 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.

5.2.5 Tape Execution

The tape entries were implemented as SVG elements. These were constructed using the **d3** API. To illustrate tape execution, we make use of the animation subpackage in d3. 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 Conclusion

6.1 Summary

The project aimed to construct a language for Turing Machines and investigate whether it is easier to learn the concept the traditional manner or through programming. To allow for the comparison of the two techniques, the project as split into 3 construction phases:

- defining the language;
- constructing a parser for the language;
- showcasing the parser in a website (the product).

The 3 phases were completed, after which there was an evaluation session aimed at comparing the teaching of Turing Machines using diagrams and programs.

The evaluation showed that the language is somewhat easier to understand and learn than the diagrammatic approach. However, due to the limited nature of the evaluation and the previous programming experience of the students, this result might not hold in general. This could likely be improved by adding some syntactic sugar to the language so that it abstracts further the details of Turing Machines.

6.2 Reflection

This project was quite an enjoyable experience for me. I was very happy to work on a self-defined project and that gave me a lot of motivation to work on it! It was also exciting connecting programming languages and Turing machines, and I found that quite exciting! This project has also taught me a lot about web development, in particular the React framework!

I do feel that due to the time constraints, I was not able to develop the language and the product completely. There are many extensions that I would have liked to add, as listed in future work. Nonetheless, I plan to work on adding these extensions during the summer!

6.3 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.

6.3.1 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.

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

A program in TML is 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 an **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 is 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.

$$\begin{aligned}
\mathit{program} &= \mathit{alphabet}\ \mathit{module}^{+}\\
\mathit{alphabet} &= \texttt{alphabet = \{}\ \mathit{seq\text{-}val}\ \texttt{\}}\\
\mathit{module} &= \texttt{module}\ \mathit{id}\ \texttt{\{}\ \mathit{block}^{+}\ \texttt{\}}\\
\mathit{block} &= \mathit{basic\text{-}block} \mid \mathit{switch\text{-}block}\\
\mathit{switch\text{-}block} &= \mathit{case\text{-}block}^{+}\\
\mathit{case\text{-}block} &= \mathit{if\text{-}block} \mid \mathit{while\text{-}block}\\
\mathit{if\text{-}block} &= \texttt{if}\ \mathit{seq\text{-}val}\ \texttt{\{}\mathit{block}^{+}\texttt{\}}\\
\mathit{while\text{-}block} &= \texttt{while}\ \mathit{seq\text{-}val}\ \texttt{\{}\ \mathit{core\text{-}com}^{+}\ \texttt{\}}\\
\mathit{basic\text{-}block} &= (\mathit{core\text{-}com} \mid \mathit{flow\text{-}com})^{+}\\
\mathit{core\text{-}com} &= \texttt{move}\ \mathit{direction} \mid \texttt{changeto}\ \mathit{value}\\
\mathit{flow\text{-}com} &= \texttt{goto}\ \mathit{id} \mid \mathit{terminate}\\
\mathit{terminate} &= \texttt{reject} \mid \texttt{accept}\\
\mathit{direction} &= \texttt{left} \mid \texttt{right}\\
\mathit{seq\text{-}val} &= (\mathit{value}\texttt{,})^{*}\ \mathit{value}\\
\mathit{value} &= \texttt{blank} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{z} \mid \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9}\\
\mathit{id} &= (\texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \ldots \mid \texttt{Z})^{+}
\end{aligned}$$

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 Σ and let T be a tape on Σ. We execute P on T inductively, as follows:

- At any point during execution, we maintain 3 objects- a tape on Σ, 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

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

    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:

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

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.

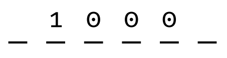
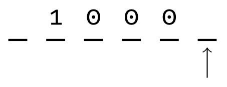


Figure A.2: *A TM tape on* $\{0, 1\}$.

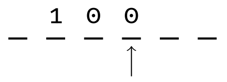
We will now illustrate this process. We execute the program `isEven` on the tape at Figure A.2.

- 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 `0` 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:



  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 `0`. So,
  - the tape value changes to `blank`;
  - 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:



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.

```
alphabet = {0, 1}
module isOdd {
   // move to the end
   while 0 {
      changeto 0
      move right
   } while 1 {
      changeto 1
      move right
   } if blank {
      changeto blank
      move left
      goto isOddCheck
   }
}
module isOddCheck {
   // accept if and only if the value is 1
   if 0, blank {
      changeto 0
      move left
      reject
```

```
    } if 1 {
        changeto blank
        move left
        accept
    }
}
```

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.

```
alphabet = {a, b}
module simple1 {
   if a, b, blank {
      changeto b
      move left
      goto simple2
   }
}
module simple2 {
   if a, b, blank {
      move right
      accept
   }
}
```

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

```
alphabet = {a, b}
module simple1 {
   if a {
      changeto b
      move left
      goto simple2
   } if b {
      changeto b
      move left
      goto simple2
   } if blank {
      changeto b
      move left
      goto simple2
   }
}
module simple2 {
   if a {
      changeto a
      move right
      accept
   } if b {
      changeto b
      move right
      accept
   } if blank {
      move right
      accept
   }
}
```

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 TML 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 σ_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 σ_n, then the next tape $T_{n+1} = T_n$. In the complete module m_n, the case for σ_n will have the command `changeto` σ_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 σ_n, then the next tapehead index $i_{n+1} = i_n - 1$. In the complete module m_n, the case for σ_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 σ_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 σ_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 σ_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 σ_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. □

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 behaviour as the TM. Later, we will construct a TM for a 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:

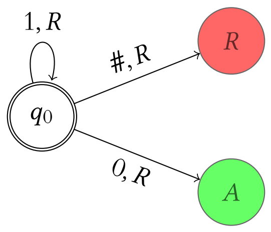


Figure B.1: *A TM that accepts binary strings containing 0*

```
alphabet = {0, 1}
module has0 {
   if 1 {
      changeto 1
      move right
      goto has0
   } if 0 {
      changeto 0
      move right
      accept
   } if blank {
      changeto blank
      move left
      reject
   }
}
```

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 σ in the alphabet Σ^+, denote $\delta(q,\sigma) = (q', \sigma', \mathtt{dir})$. We add an *if* case in the *switch* command corresponding to letter σ with the following commands:
  - `changeto` σ'
  - `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 Σ;
- 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 TML 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 σ_n be the letter at index $i_n = j_n$ on the tape $T_n = S_n$. Denote $q(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \texttt{dir})$. In that case,

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

  and the next state is q_{n+1}.
  - We know that the module m_n in TML program P corresponds to the state q_n, so it has a `changeto` σ_{n+1} command for the case σ_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:

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

    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 σ_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. □

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

- the value σ' 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 σ 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 Σ, the transition function δ and initial state q_0 corresponding to the first module in P is the **corresponding TM for** P.

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

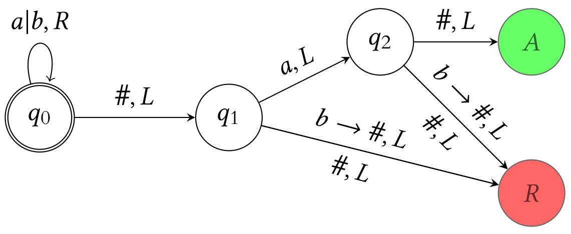


Figure B.2: The TM corresponding to the program above. The state q_0 corresponds to the module `moveToEnd`*; the state q_1 corresponds to the module* `checkAFirst`*; and the state q_2 corresponds to the module* `checkASecond`*.*

```
alphabet = {a, b}
module moveToEnd {
   while a {
      changeto a
      move right
   } while b {
      changeto b
      move right
   } if blank {
      changeto blank
      move left
      goto checkAFirst
   }
}
module checkAFirst {
   if a {
      changeto blank
      move left
      goto checkASecond
   } if b, blank {
      changeto blank
      move left
      reject
   }
}
module checkASecond {
   if a {
      changeto blank
      move left
      accept
   } if b, blank {
      changeto blank
      move left
      reject
   }
}
```

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

Theorem 3. *Let P be a TML 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 TML 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. Without loss of generality, assume that P is complete. 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 σ_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 σ_n is a *while* block, then we know that its body is partially complete, and so is composed of the following commands:
    - `changeto` σ_{n+1}
    - `move dir`

    So, we have $\delta(q_n, \sigma_n) = (q_n, \sigma_{n+1}, \texttt{dir})$. 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` σ_{n+1}
    - `move dir`
    - `accept`, `reject` or `goto` m_{n+1}.

    So, we have $\delta(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \texttt{dir})$, 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

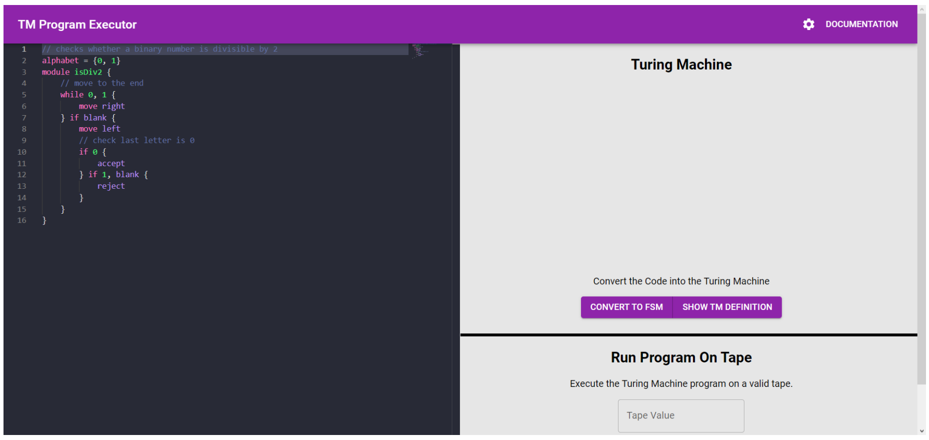
Since TML programs and TMs are equivalent, this implies that TML programs are a model for computation. Note that while we have given a proof for equivalence for TMs, this representation is based on accepting and rejecting programs only. Not all TMs are of this form. In particular, it is also possible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



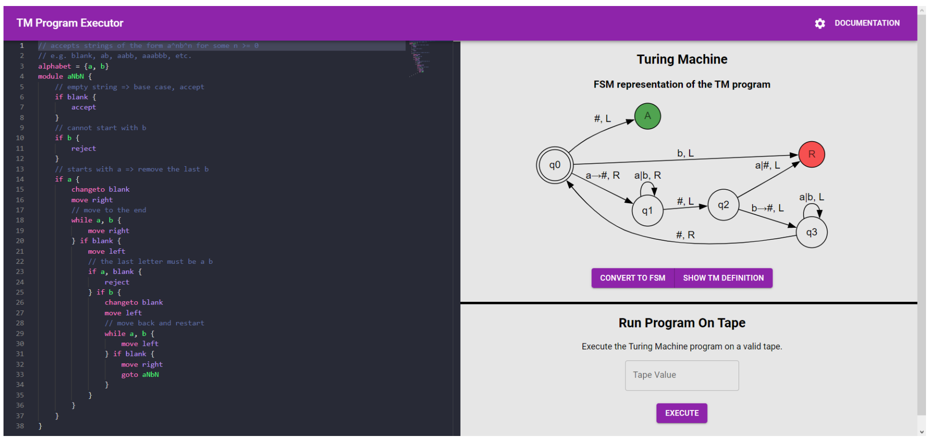
```
// checks whether a binary number is divisible by 2
alphabet = {0, 1}
module isDiv2 {
    // move to the end
    while 0, 1 {
        move right
    } if blank {
        move left
        // check last letter is 0
        if 0 {
            accept
        } if 1, blank {
            reject
        }
    }
}
```

Convert the Code into the Turing Machine

Execute the Turing Machine program on a valid tape.

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

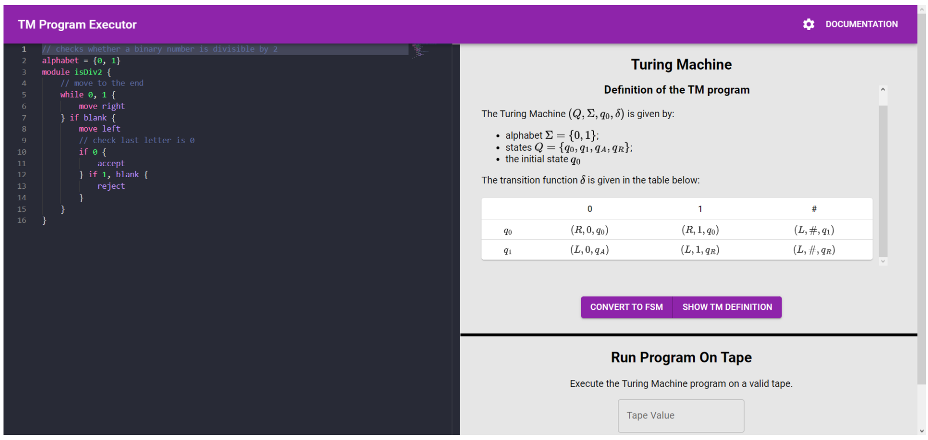
The initial rendering of the homepage is given in Figure C.1. It shows an example program initially, with no TM conversion or tape execution. We can change the program in settings. The user can press the button in TM panel to convert the program to either the FSM representation or the definition version.



```
// accepts strings of the form a^nb^n for some n >= 0
// e.g. blank, ab, aabb, aaabbb, etc.
alphabet = {a, b}
module aNbN {
    // empty string => base case, accept
    if blank {
        accept
    }
    // cannot start with b
    if b {
        reject
    }
    // starts with a => remove the last b
    if a {
        changeto blank
        move right
        // move to the end
        while a, b {
            move right
        } if blank {
            move left
            // the last letter must be a b
            if a, blank {
                reject
            } if b {
                changeto blank
                move left
                // move back and restart
                while a, b {
                    move left
                } if blank {
                    move right
                    goto aNbN
                }
            }
        }
    }
}
```

Execute the Turing Machine program on a valid tape.

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



```
// checks whether a binary number is divisible by 2
alphabet = {0, 1}
module isDiv2 {
    // move to the end
    while 0, 1 {
        move right
    } if blank {
        move left
        // check last letter is 0
        if 0 {
            accept
        } if 1, blank {
            reject
        }
    }
}
```

Turing Machine

Definition of the TM program

The Turing Machine (Q, Σ, q_0, δ) is given by:

- alphabet $\Sigma = \{0, 1\}$;
- states $Q = \{q_0, q_1, q_A, q_R\}$;
- the initial state q_0

The transition function δ is given in the table below:

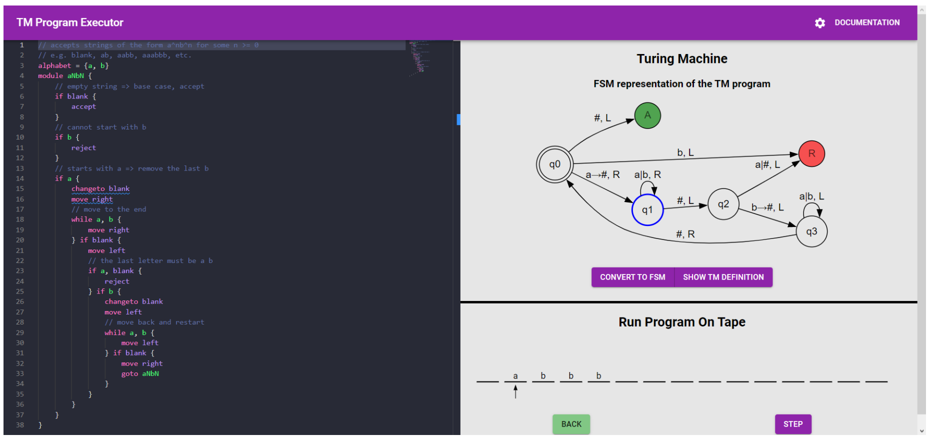
	0	1	$\#$
q_0	$(R, 0, q_0)$	$(R, 1, q_0)$	$(L, \#, q_1)$
q_1	$(L, 0, q_A)$	$(L, 1, q_R)$	$(L, \#, q_R)$

Run Program On Tape

Execute the Turing Machine program on a valid tape.

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.

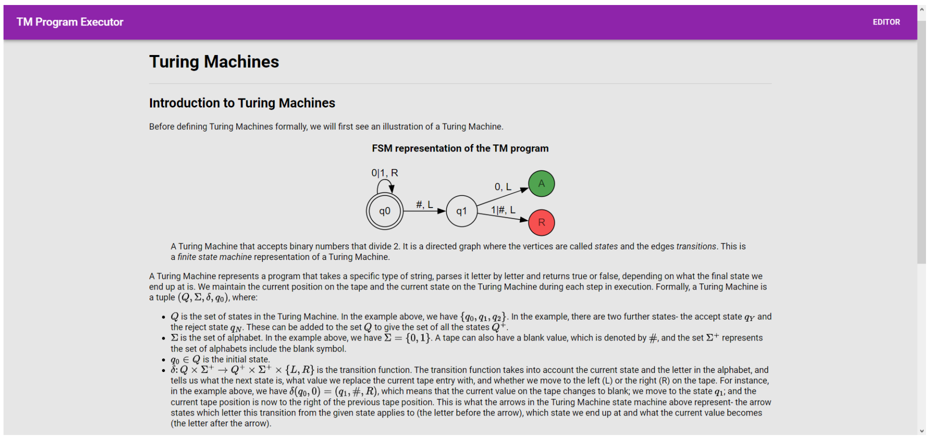


```
// accepts strings of the form a^nb^n for some n >= 0
// e.g. blank, ab, aabb, aaabbb, etc.
alphabet = {a, b}
module aNbN {
    // empty string => base case, accept
    if blank {
        accept
    }
    // cannot start with b
    if b {
        reject
    }
    // starts with a => remove the last b
    if a {
        changeto blank
        move right
        // move to the end
        while a, b {
            move right
        } if blank {
            move left
            // the last letter must be a b
            if a, blank {
                reject
            } if b {
                changeto blank
                move left
                // move back and restart
                while a, b {
                    move left
                } if blank {
                    move right
                    goto aNbN
                }
            }
        }
    }
}
```

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



Turing Machines

Introduction to Turing Machines

Before defining Turing Machines formally, we will first see an illustration of a Turing Machine.

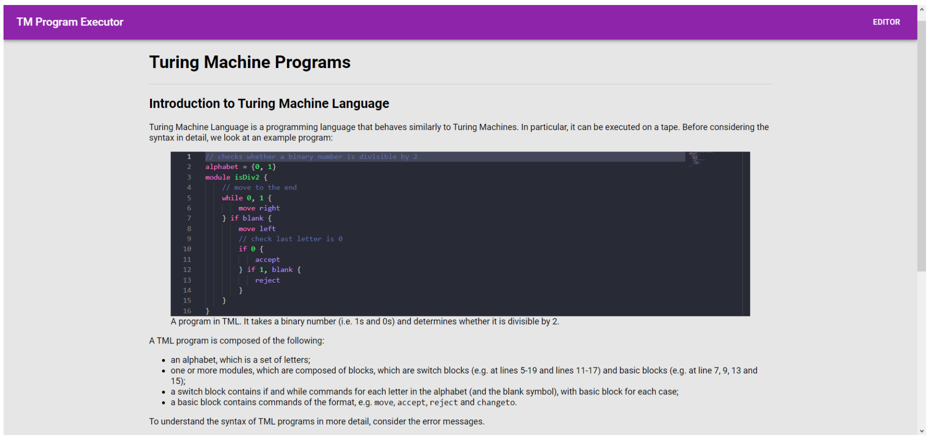
FSM representation of the TM program

A Turing Machine that accepts binary numbers that divide 2. It is a directed graph where the vertices are called *states* and the edges *transitions*. This is a *finite state machine* representation of a Turing Machine.

A Turing Machine represents a program that takes a specific type of string, parses it letter by letter and returns true or false, depending on what the final state we end up at is. We maintain the current position on the tape and the current state on the Turing Machine during each step in execution. Formally, a Turing Machine is a tuple (Q, Σ, δ, q_0), where:

- Q is the set of states in the Turing Machine. In the example above, we have $\{q_0, q_1, q_2\}$. In the example, there are two further states- the accept state q_Y and the reject state q_N. These can be added to the set Q to give the set of all the states Q^+.
- Σ is the set of alphabet. In the example above, we have $\Sigma = \{0, 1\}$. A tape can also have a blank value, which is denoted by $\#$, and the set Σ^+ represents the set of alphabets include the blank symbol.
- $q_0 \in Q$ is the initial state.
- $\delta: Q \times \Sigma^+ \to Q^+ \times \Sigma^+ \times \{L, R\}$ is the transition function. The transition function takes into account the current state and the letter in the alphabet, and tells us what the next state is, what value we replace the current tape entry with, and whether we move to the left (L) or the right (R) on the tape. For instance, in the example above, we have $\delta(q_0, 0) = (q_1, \#, R)$, which means that the current value on the tape changes to blank; we move to the state q_1; and the current tape position is now to the right of the previous tape position. This is what the arrows in the Turing Machine state machine above represent- the arrow states which letter this transition from the given state applies to (the letter before the arrow), which state we end up at and what the current value becomes (the letter after the arrow).

Figure C.5: The documentation webpage for TM



Turing Machine Programs

Introduction to Turing Machine Language

Turing Machine Language is a programming language that behaves similarly to Turing Machines. In particular, it can be executed on a tape. Before considering the syntax in detail, we look at an example 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. It takes a binary number (i.e. 1s and 0s) and determines whether it is divisible by 2.

A TML program is composed of the following:

- an alphabet, which is a set of letters;
- one or more modules, which are composed of blocks, which are switch blocks (e.g. at lines 5-19 and lines 11-17) and basic blocks (e.g. at line 7, 9, 13 and 15);
- a switch block contains if and while commands for each letter in the alphabet (and the blank symbol), with basic block for each case;
- a basic block contains commands of the format, e.g. `move`, `accept`, `reject` and `changeto`.

To understand the syntax of TML programs in more detail, consider the error messages.

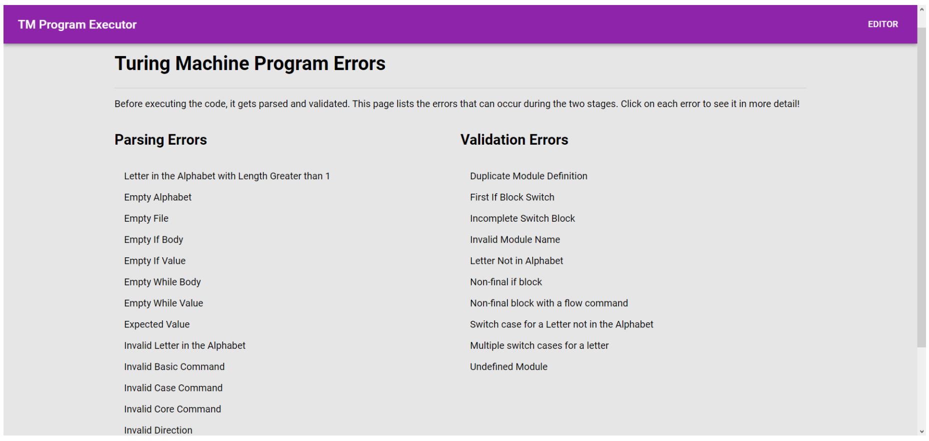
Figure C.6: The documentation webpage for TML

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.



Turing Machine Program Errors

Before executing the code, it gets parsed and validated. This page lists the errors that can occur during the two stages. Click on each error to see it in more detail!

Parsing Errors

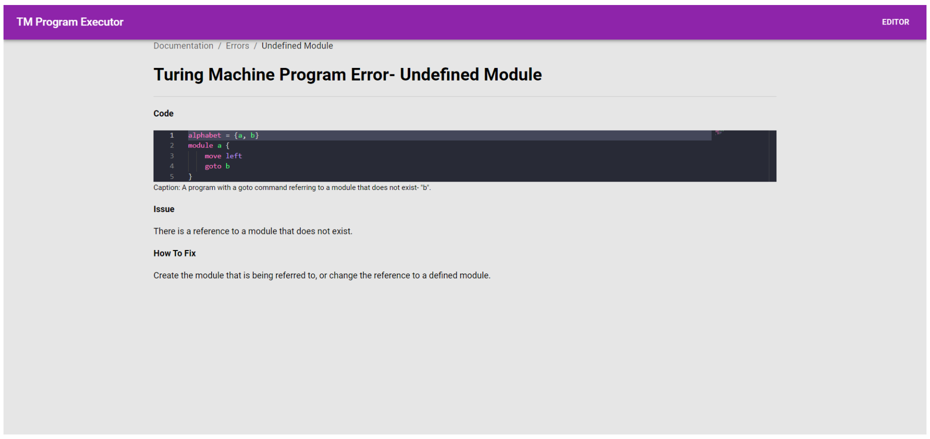
- Letter in the Alphabet with Length Greater than 1
- Empty Alphabet
- Empty File
- Empty If Body
- Empty If Value
- Empty While Body
- Empty While Value
- Expected Value
- Invalid Letter in the Alphabet
- Invalid Basic Command
- Invalid Case Command
- Invalid Core Command
- Invalid Direction

Validation Errors

- Duplicate Module Definition
- First If Block Switch
- Incomplete Switch Block
- Invalid Module Name
- Letter Not in Alphabet
- Non-final if block
- Non-final block with a flow command
- Switch case for a Letter not in the Alphabet
- Multiple switch cases for a letter
- Undefined Module

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.



Turing Machine Program Error- Undefined Module

Code

```
alphabet = {a, b}
module a {
    move left
    goto b
}
```

Caption: A program with a goto command referring to a module that does not exist- "b".

Issue

There is a reference to a module that does not exist.

How To Fix

Create the module that is being referred to, or change the reference to a defined module.

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:

```
// checks whether a binary number is divisible by 2
alphabet = {0, 1}
module isDiv2 {
   // move to the end
   while 0, 1 {
      move right
   } if blank {
      move left
      // check last letter is 0
      if 0 {
         accept
      } if 1, blank {
         reject
      }
   }
}
```

- isDiv2Rec:

```
// checks whether a binary number is divisible by 2, recursively
alphabet = {0, 1}
module isDiv2Rec {
   // recursive case: not at the end => move closer to the end
   if 0, 1 {
      move right
      goto isDiv2Rec
   }
   // base case: at the end => check final letter 0
   if blank {
      move left
      if 0 {
         accept
      } if 1, blank {
         reject
      }
   }
}
```

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

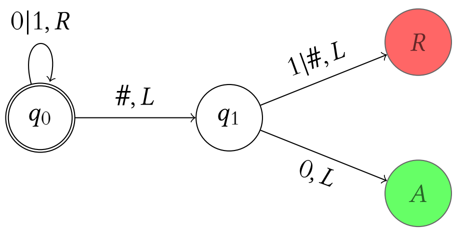


Figure D.1: The TM for `isDiv2` and `isDiv2Rec`.

- `aNbN`:

```
// accepts strings of the form a^nb^n for some n >= 0
// e.g. blank, ab, aabb, aaabbb, etc.
alphabet = {a, b}
module aNbN {
   // empty string => base case, accept
   if blank {
      accept
   }
   // cannot start with b
   if b {
      reject
   }
   // starts with a => remove the last b
   if a {
      changeto blank
      move right
      // move to the end
      while a, b {
         move right
      } if blank {
         move left
         // the last letter must be a b
         if a, blank {
            reject
         } if b {
            changeto blank
            move left
            // move back and restart
            while a, b {
               move left
            } if blank {
               move right
               goto aNbN
            }
         }
      }
   }
}
```

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

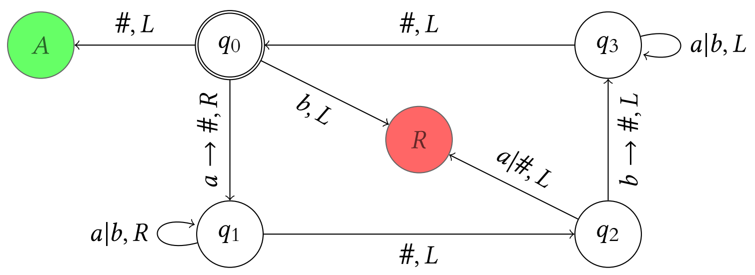


Figure D.2: The TM for `aNbN`.

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:

```
alphabet = {0, 1}
module mystery1 {
   while 0, 1 {
      move right
   } if blank {
      move left
      if blank, 0 {
         reject
      } if 1 {
         move left
         if blank, 1 {
            reject
         } if 0 {
            accept
         }
      }
   }
}
```

   (a) Does the program accept the values:
      i. 10 (NOTE: This is 2 in decimal)
      ii. 1
      iii. 100 (NOTE: This is 4 in decimal)
      iv. 101 (NOTE: This is 5 in decimal)
      v. 110 (NOTE: This is 6 in decimal)
   (b) Describe the values this program accepts.

2. Consider the following TML program:

```
alphabet = {a, b}
module mystery2 {
   if blank {
      accept
   } if b {
      reject
   } if a {
      changeto blank
      move right
      if b, blank {
         reject
      } if a {
         changeto blank
         move right
         while a, b {
            move right
         } if blank {
            move left
            if a, blank {
               reject
            } if b {
               changeto blank
               move left
               if a, blank {
                  reject
               } if b {
                  changeto blank
                  move left
                  while a, b {
                     move left
                  } if blank {
                     move right
                     goto mystery2
                  }
               }
            }
         }
      }
   }
}
```

(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!).

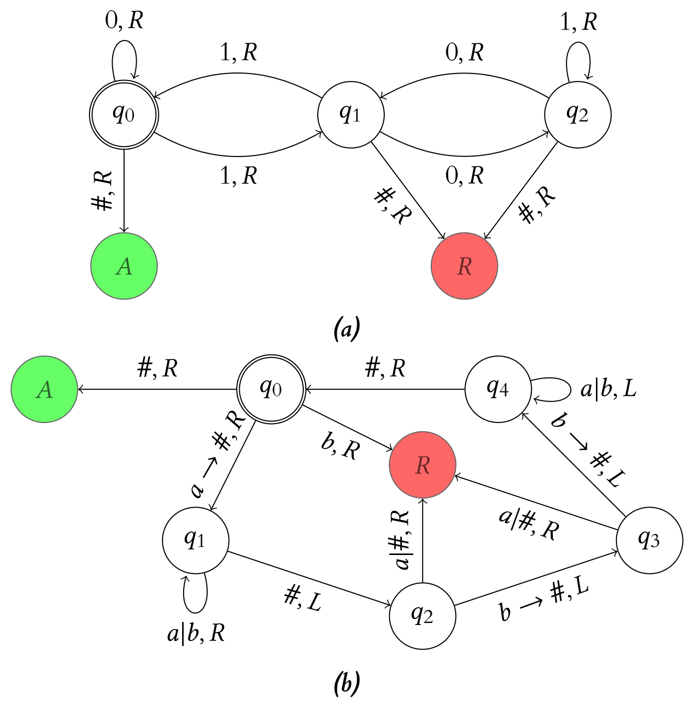


Figure D.3: 2 Turing Machines

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 3rd/4th/5th 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) 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 Gautam

Student Number 2481471G

Student's Signature Pete

Supervisor's Signature [illegible]

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	○	○	○	○
TML is easy to write programs in	○	○	○	○
I was able to fix errors in my code using the error messages provided	○	○	○	○
I was able to easily reason executing a program on a tape	○	○	○	○

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	○	○	○	○
I find it easier to reason what a TML program accepts than a TM	○	○	○	○

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	○	○	○	○
The presentation of the website is intuitive	○	○	○	○
There were no visible bugs in the website	○	○	○	○
The website was fast	○	○	○	○
The website feels complete	○	○	○	○
The code execution was easy to follow	○	○	○	○
The code editor was easy to use	○	○	○	○

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?

Bibliography

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

AngularJS (2023), 'Angularjs'. Accessed 2023/03/15.
URL: *https://angular.io*

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.

Google (2023), 'Material design'. Accessed 2023/03/15.
URL: *https://m3.material.io*

Graphviz (2023*a*), 'DOT language'. Accessed 2023/03/15.
URL: *https://graphviz.org/doc/info/lang.html*

Graphviz (2023*b*), 'Graphviz'. Accessed 2023/03/15.
URL: *https://graphviz.org*

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

Microsoft (2023), 'Monaco editor'. Accessed 2023/03/15.
URL: *https://microsoft.github.io/monaco-editor/*

ReactJS (2023), 'Reactjs'. Accessed 2023/03/15.
URL: *https://reactjs.org*

Stack Exchange (2022), 'Stack overflow developer survey 2022'. Accessed 2023/03/15.
URL: *https://survey.stackoverflow.co/2022/*

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

w3techs (2023), 'Comparison of the usage statistics of React vs. Angular for websites'. Accessed 2023/03/15.
URL: *https://w3techs.com/technologies/comparison/js-angularjs,js-react*

Webpack (2023), 'Webpack'. Accessed 2023/03/15.
URL: *https://webpack.js.org*