University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Turing Machine Language

Pete Gautam

March 7, 2023

Abstract

Acknowledgements

Education Use Consent

Contents

1 Introduction

1.1 Motivation

Turing Machine (TM) is 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 (FSM). 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- defining the language, creating the parser for the language and finally creating a website that allows a user to make use of the parser.

The first aim is to define a programming language to represent TMs, called the Turing Machine Language (TML). A TM can be executed on a (valid) tape, and this is what programs in TML will try to simulate. Hence, the language should be able to represent all the operations that a TM supports.

Next, a parser will be created for TML. 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 TM; and
- executed on a valid tape (string).

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

1.3 Summary

TODO

2 Background

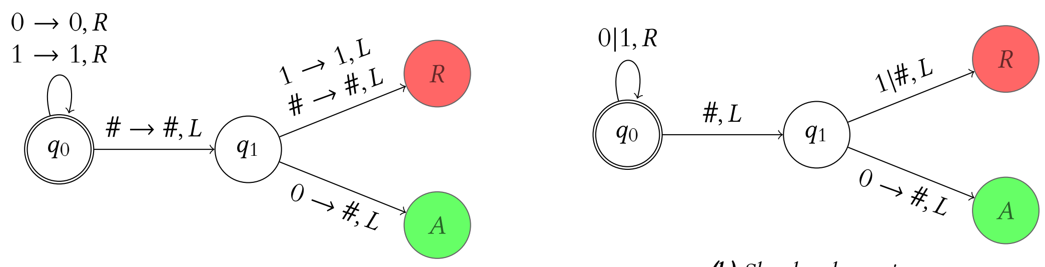
2.1 Turing Machine

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 et al. (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 FSM- subfigure (a) shows how all the transitions can be specified, while subfigure (b) only shows the transitions where the tapehead value is getting changed. We will make use of the shorthand FSM representation from now.

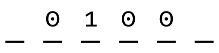
2.1.2 Executing a TM on a tape

Let Σ be an alphabet. A *tape T on Σ* is a function $T \colon \mathbb{Z} \to \Sigma^+$. In particular, the tape has infinite entries in both directions. Moreover, T is a *valid tape* if only finitely many symbols on T are not `blank`, and all the values that can be non-`blank` are non-`blank`. That is, there exist integers a, b such that for all $x \in \mathbb{Z}$, $T(x)$ is not `blank` if and only $x \geq a$ and $x \leq b$.

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

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

Then, the following figure represents the tape T:



We will assume that the first non-blank value is at index 0.

A TM can be executed on a tape. Let M be a TM with alphabet Σ, and let T be a (valid) tape on Σ. We execute M on T inductively, as follows:

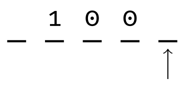
- At any point during execution, we maintain 3 objects- a tape on Σ, a state in M and an index in the tape (called the *tapehead index*).
- At the start, the tape is T; the tapehead index is 0; and the state is the initial state q_0.
- At some point during the execution, assume that we have the tape 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', and the next tape S' and the next tapehead index j' are given by:

  $$S'(x) = \begin{cases} t' & x = i \\ S(x) & \text{otherwise,} \end{cases} \qquad j' = \begin{cases} j + 1 & \texttt{dir} = \texttt{right} \\ j - 1 & \texttt{dir} = \texttt{left}. \end{cases}$$

  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'.

We illustrate this process with the TM in Figure 2.1. First, a tape on the TM will only have values 0 and 1, and can be thought of as a binary number. So, we can execute the TM on the tape `100`. In that case, the execution proceeds as follows:

- We start at q_0 with the tapehead value the first entry, i.e. `1`.
- Since the value is `1`, we stay at q_0 and move the tapehead pointer to the right. Now, the tapehead value is `0`.
- The transition for `0` and `1` are the same with respect to q_0, so we keep moving to the right until we end up at the blank symbol. At this point, the tape is as follows:



- Now, since the tapehead value is `blank`, we move to the left and the current state becomes q_1.
- The current value is a `0`, so we move to the accept state A. Hence, the tape gets accepted.

As we can see, the state q_0 is used to traverse to the first blank symbol (i.e. the end of the string). At that point, we move to the state q_1. At this point, we accept the string if and only if the current tapehead value (which is the last entry on the tape) 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

TODO

2.2 Parser

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

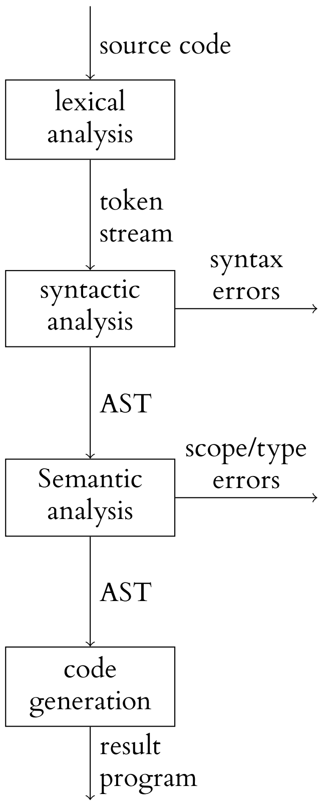


Figure 2.2: *The data flow between the compilation phases.*

We will now consider the different phases of the compilation process. This is summarised in Figure 2.2. This figure, along with most of the content in this section, has been adapted from Aho et al. (2007).

2.2.1 Lexical Analysis

First, we perform *lexical analysis* using a *lexer*.

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 *tokens*, which are a sequence of words in source code along with labels. For instance, consider the mathematical expression `1 + 2`. We can convert this expression into 3 tokens: `(1, NUM)`, `(+, PLUS)` and `(2, NUM)`.

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 a syntax tree. This will be detected during this phase, at which point we can throw a syntax error.

A syntax tree represents the program as a tree of nodes. Typically, the internal nodes represent operations and the leaves represent their arguments. An AST is a compact representation of a syntax tree that does not feature all the nodes. The abstract syntax tree for the expression `(1 + 2) * (3 + 4)` is given in Figure 2.3.

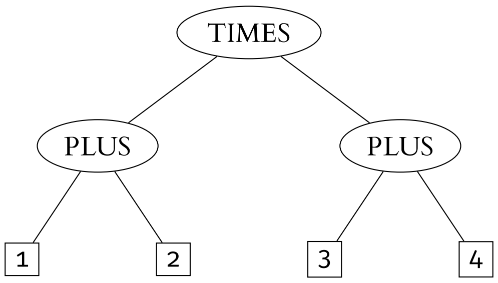


Figure 2.3: *The AST for the expression* `(1 + 2) * (3 + 4)`

There are many ways to parse the stream of tokens. A common method is *recursive-descent* parsing. Here, we recursively parse the source code and generate nodes within the AST. For instance, we initially start by parsing a program. If we then encounter an expression, and this is allowed in the grammar of the language, we parse an expression.

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, e.g. if we see the token `if`, then we know we are parsing an if command. Such a parser makes use of the `match` method that is used to match the next token value.

A snippet of a recursive-descent class `CodeParser` is given below, which illustrates how an if statement might be parsed.

```
class CodeParser {
   parseIf():IfContext {
      match("if");
      var condition = parseExpr();

      match("then");
      var expr1 = parseExpr();

      match("else");
      var expr2 = parseExpr();

      return new IfContext(condition, expr1, expr2);
   }

   // ... other parsing methods
}
```

2.2.3 Semantic Analysis

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

In type errors, we check whether the AST has some type mismatch, e.g. `1 + true`. If there are no errors, then we will have assigned types to each variable. This information will likely be used during code generation.

In scope errors, we check whether some identifier present in code is undefined. To do so, we need to keep track of all the variables that are in scope at this point. 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.

There are many ways to traverse the AST. One way of doing so makes use of the visitor design pattern (Gamma et al. (1995)). Here, every node in the AST is represent by some child class of the `Context` class, e.g. `IfContext` for if statements. The `Context` class has an abstract method `accept` that allows us to visit different nodes. We can then make use of this when traversing the AST.

We illustrate the visitor design pattern with an example. So, assume that we have the following class to represent if statements.

```
class IfContext extends Context {
   accept<T>(validator:CodeVisitor<T>):T? {
      return validator.visitIf(this);
   }

   // ... relevant methods for if context
}
```

In that case, we can define the following method to 'visit' an if statement and validate it.

```
class TypeValidator implements CodeVisitor<Type> {
   visit(context:Context):Type? {
      return context.accept(this);
   }

   // validate that the condition is of type bool
   visitIf(ifContext:IfContext):Type? {
      var type = visit(ifContext.condition);
      if (type != Type.BOOL) {
         throw new TypeError("Expected if condition to be of type bool.");
      }

      return undefined;
   }

   // ... other validator methods
}
```

To run the `TypeValidator`, we visit the entire program.

The advantage of using the visitor design pattern is that we can traverse all the nodes without altering any of the `Context` classes; we can just construct another `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.

2.2.4 Code Generation

Finally, we convert the AST into code in the target language. In particular, we traverse the tree left-to-right and convert each phrase from the source language to the target. This can also be done using the visitor design pattern. Here, the return type is expected to be a representation of the target language.

3 Requirements

I used MoScoWs to specify the requirements for the project. In particular, the requirements were partitioned into one of the 4 sections: must have, should have, could have and will not have. Since the project has multiple parts, each part had its own MoScoW section.

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;
- how to execute a program on a valid tape; and
- a proof of equivalence between TMs and TML programs (which involves constructing a TM for a TML program, and vice versa).

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.

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.

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- Website

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.

Should Have The code editor should support syntax highlighting. Assuming that the initial FSM representation rendered by the website places the states in a random manner, without considering which states are linked and should be placed closer, the user should be able to drag them around.

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 Turing Machine Language (TML) should be a language equivalent to TMs in tape execution. That is, for any TM that can be run on some tape, there should exist a TML program that can run on the same tape exactly in the same manner.

For TML to replicate TMs, it must support the following 3 operations:

- we can change the tapehead value to some letter in the alphabet;
- we can move the tapehead pointer left or right; and
- we can move from one state to another (including the accept and the reject state).

The first two operations are expected to be directly supported. The third operation might be implicitly supported since the language need not have the concept of a state.

In TMs, the transition function depends on tapehead value and the current state. We expect this to be supported within the TML. Since the TML might not support states, this should be done in a way that combines well with the 3 main operations of a TM.

It is important that TML is not just another representation for TMs. The aim of the TML is that it is equivalent to TMs, but more closely resembles a traditional programming language (PL) than a TM. For this reason, the language should follow syntax that is common in traditional PLs. It should also ensure that the details of TM are abstracted- it is meant to be an intermediate representation of a TM.

4.2 Parser

The parser should take a program in TML and convert it into a TM or execute it on a tape.

It should first perform lexical analysis that produces a stream of tokens. However, since we expect the language to be quite simple, this stage might be unnecessary- it should be fine to just use a stream of source code phrases.

Next, the stream of tokens should be parsed into an AST. Although there are many ways to implement the parser, the parser should be able to correctly identify any syntax errors, and give clear and succinct error messages.

There are many frameworks that can perform lexical and syntactic analysis given a grammar definition for the language. It might be beneficial to use these since they are more likely to be correct and robust. Moreover, for complex languages, it can be quite difficult and time-consuming to construct a correct parser by hand. The TML is expected to be quite simple, so this might not be a relevant issue.

After the AST has been constructed, semantic analysis should be performed. Since the language is expected to be simple, it will likely not make use of a type system. Hence, there should be no need to do type checking. On the other hand, it is expected that the language makes use of identifiers, e.g. to identify the equivalent of states for transition. Hence, it is expected that there is some scope checking in this stage. Also, like with a FSM representation of TM, we should also ensure that the transition data is complete (i.e. we specify where we transition to for each letter in the alphabet).

Finally, the AST should be used to generation code. There are 2 things to be done here- converting a program into a TM and executing it. To do so, we would need to have some representation of a TM. It is likely a good idea to use the formal definition instead of a directed graph representation- the formal definition should make execution much easier to implement.

It is possible to avoid defining execution and just rely on execution of a TM (and hence create a true compiler). However, it might be more optimal to define execution directly on TML programs. In particular, the language might provide shortcuts that TM does not have which could make execution more efficient.

It is important that the parser be written in a way that is compatible with web deployment- the parser will be used in the product. Moreover, since the website is expected to have live syntax highlighting, the error messages should help the user fix any bugs in their code.

4.3 Product

The website should allow the parser to be used. In particular, it should allow the user to type in a program and then:

- convert it into a TM; or
- run it on a valid tape, one step at a time.

5 Implementation

5.1 Language

The TML has been implemented 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 value in some direction;
- it makes use of `changeto` commands to change the tapehead value to some letter in the alphabet.

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

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

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

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

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

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

With these constructs, we have constructed a language that is equivalent to TMs.

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

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

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

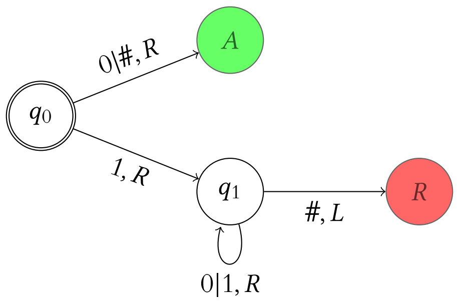


Figure 5.1: A TM with a self-loop at the state q_1

Although nesting has made the language more like a typical PL, there is still one issue- a self-loop in a non-starting state. To see this, we consider the TM at Figure 5.1. 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- recursion would convert the self-loop of q_1 to a transition from q_1 to q_0. It is always possible to represent a TM as a complete program, but this representation is very close to a TM. We want there to be another way to represent this program that resembles a PL better.

To allow for self-loops using nesting, we introduce a new construct- a `while` case. This is similar to an `if` case, but after the block is executed, we stay at the same block (which does not necessarily mean that the same case is run). This is precisely a self-loop. We can now convert the TM above 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 code execution. The proof of equivalence is composed of several proofs, which involve:

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

5.2 Parser

TODO

5.3 Product

TODO

6 Evaluation

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

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

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

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

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

During the evaluation session, students were first introduced to TML programs. They were then expected to understand what language two mystery TML programs accept. This was done by checking whether the programs accepted some values, which should have helped them understand the way the program operates and decode the language it accepts. They were expected to use the product to help them follow the code and understand the steps in execution.

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

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

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

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

6.1 Evaluation Results

6.1.1 Parser and Language Evaluation

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

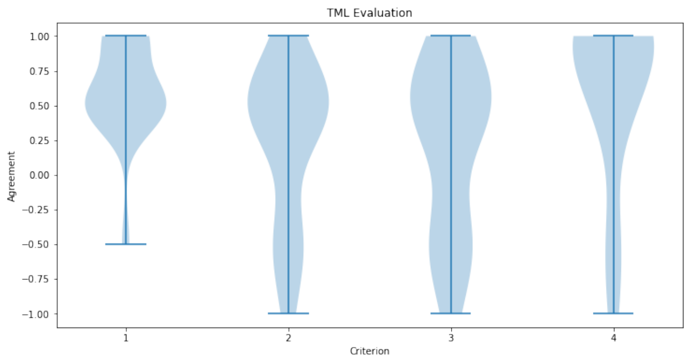


Figure 6.1: *A violin plot that summarises the results of the survey relating to the TML with respect to the 4 criteria. The agreement value refers to how much the user agrees to the statement:* -1 *is strongly disagree;* -0.5 *disagree;* 0.5 *agree and* 1 *strongly agree. The whiskers show the range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

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

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

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

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

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

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

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

```
if a {
   changeto blank
   goto someModule
}
```

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

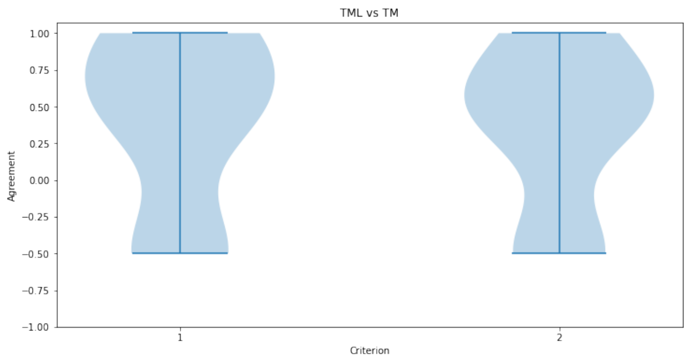


Figure 6.2: A violin plot that summarises the results of the survey that compare TM and TML with respect to the 2 criteria.

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

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

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

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

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

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

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

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

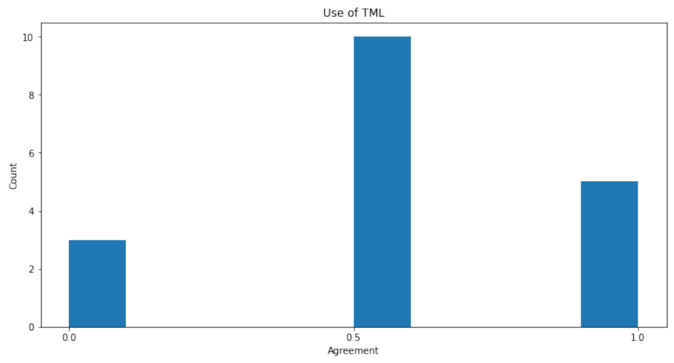


Figure 6.3: *A histogram that summarises whether the users would consider writing a TML program before drawing a TM. 0 means no; 0.5 means maybe; and 1 means yes.*

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

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

6.1.2 Product Evaluation

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

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

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

The results of the survey are summarised in Figure 6.4.

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

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

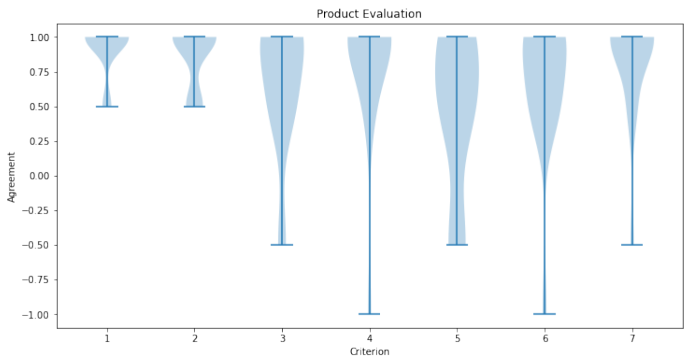


Figure 6.4: *A violin plot that summarises the results of the survey relating to the website with respect to the 7 criteria.*

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

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

6.2 Limitations to Evaluation

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

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

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

7 Conclusion

7.1 Summary

TODO

7.2 Reflection

TODO

7.3 Future Work

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

7.3.1 Language

Although the language is equivalent to TMs, 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) in one command, e.g. `move end`; and
2. an else clause (an if block for the remaining characters);
3. the ability for modules to be parametrised with respect to letters.

We illustrate these issues with the following program:

```
// accept strings that start and end with opposite letters
alphabet={a, b}
module oppositeLetters {
   if a {
      move right
      while a, b {
         move right
      } if blank {
         move left
         if b {
            accept
         } if a, blank {
            reject
         }
      }
   } if b {
      move right
      while a, b {
         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, which is relatively simple, we are traversing to the end twice- at lines 5-8 and 17-21. This issue was brought up plenty of times during the evaluation sessions and was also given as an example of a feature to add in the survey. It would be a highly beneficial feature to add and should be quite a simple extension.

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, 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. Hence, this would be a great feature to be added.

The final feature is quite interesting. There are many programs where some modules are very similar and only differ in a character. In particular, the program above has 2 essentially mirrored blocks at lines 4-16 and 16-28. In fact, 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. So, I believe this would be a useful feature, and would get rid of many duplicate/similar code within programs! Adding this feature would also serve as a benefit to TMs since this issue also exists in TMs.

Finally, there is an open question about the language- the ability to convert a TM into a 'good' TML program. Currently, the proof of equivalence makes use of an algorithm to convert a valid TML program to a complete TML program, but not vice versa. Complete TML programs can be thought of as another representation for TMs, and so add no further benefit to the language. It is the ability to nest statements within if blocks that really makes the language rich and different to TMs. For this reason, I believe it would be a good idea to devise an algorithm to convert a TM (or a complete TML program) to a nested TML program.

7.3.2 Product

There are many possible improvements to the website, which were discovered during production and user evaluation. These were 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 hundreds of times;
- the ability to collapse the panels; and
- the ability to customise the number of tape entries to be 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 more usable. They would also make it easier to keep track of code execution.

Another feature that can be added is to make the website more responsive. The FSM is currently being produced using the graphviz framework. The resulting image has hard-coded dimensions which makes it hard to make the panel responsive. It is not completely possible to make the graph fixed; only the maximum size can be specified. Moreover, the constraints are not taken into consideration when the FSM is produced. This means that for complex FSMs with about 20 nodes, the nodes are quite small so it is hard to follow execution. To fix this, one of the following features could be added:

- the ability to zoom in and drag the FSM panel; or
- the ability to scroll the FSM panel.

A | TML Specification

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

Consider the following TML program.

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

A program in TML will be used to execute on a tape, so the syntax used guides us in executing the program on a tape. We will see that later. For now, we consider the rules of the TML program:

- A valid TML *program* is composed of the *alphabet*, followed by one or more *modules*. In the example above, the alphabet of the program is $\{0, 1\}$, and the program has a single module called `isEven`.
- A module contains one or more *blocks* (a specific sequence of commands). There are two types of blocks- *basic blocks* and *switch blocks*.
- A basic block consists of *basic commands* (*changeto*, *move* or *flow* command). A basic block consists of at least one basic command, but it is not necessary for a basic block to be composed of all the basic commands. If multiple commands are present in a basic block, they must be in the following order- *changeto*, *move* and *flow* command. In the program above, there 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.

The EBNF of the TML is given below:

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

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 $i' = i - 1$.

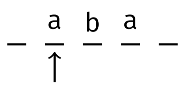
- we either terminate or determine the next block b' to execute (in decreasing precedence):
  - if the block is the body of a while case block, then the next block $b' = b$, i.e. we execute this switch block again (not necessarily the same case block);
  - if the block contains a terminating *flow* command, 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.

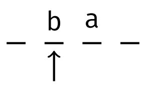
We now illustrate the execution process. So, consider the following TML program:

```
// accepts strings that are equal to their reverse
alphabet = {a, b}
module palindrome {
   // base case: accept
   if blank {
      accept
   }
   // a: remove it and check the final character is a
   if a {
      changeto blank
      move right
      // move to the end
      while a, b {
         move right
      } if blank {
         move left
         // restart if a (or blank)
         if blank, a {
            changeto blank
            move left
            goto restart
         }
         // ends in b => reject
         if b {
            reject
         }
      }
   }
   // b: remove it and check the final character is b
   if b {
      changeto blank
      move right
      // move to the end
      while a, b {
         move right
      } if blank {
         move left
         // restart if b (or blank)
         if blank, b {
            changeto blank
41              move left
42              goto restart
43           }
44           // ends in a => accept
45           if a {
46              reject
47           }
48        }
49     }
50  }
51  // go to the start and restart execution
52  module restart {
53     while a, b {
54        move left
55     } if blank {
56        move right
57        goto palindrome
58     }
59  }
```

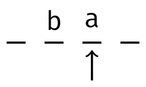
We will execute the program on the following tape.



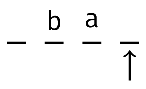
The arrow points at the tapehead value. We first execute the first block in the module `palindrome`. Since the tapehead value is `a`, we execute the basic block at lines 5-20. So, we change the tapehead value to `blank`, and the tapehead moves to the right by one step. Since this is an *if*-block, without a flow command, and there is a block following this one, the next block to be executed is the switch block at lines 8-19. Now, the current tape is the following.



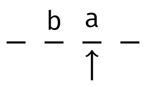
The current block is a switch block. The tapehead value is `b`, so we are at the *while* command at line 9. The basic block here only contains a *move* command. So, we leave the tape as is, and the tapehead moves to the right once. This is a *while* command, so the next block to execute is still this switch block. The current tape state is the following.



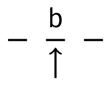
The current block is still a switch block. The tapehead value is `a`, so we execute the same *while* command at line 9. Moreover, the next block to execute is still the switch block. Now, the current tape state is the following.



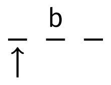
For the third time, we are executing the same switch block. Now, however, the tapehead value is `blank`, so we execute the first block of the *if* command at line 5, i.e. we move to the left. Since this is an *if* command and this is not the last block in the if command, the next block to execute is the switch block at lines 12-18.



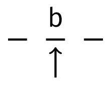
Now, the current block is a switch block. The tapehead value is `a`, so we take the basic block at lines 12-16. The value of the tapehead becomes `blank`, and moves to the left. The next block to execute is the switch block in `restart`.



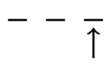
Since the current tapehead value is `b`, we execute the basic block at line 39. So, we move to the left, and the tape is left as is. Moreover, since this is a while block, the next block to execute is still the switch block.



Since the current tapehead state is `blank`, we execute the basic block at lines 40-43. So, we move to the left, and the tape is left as is. The next block to execute is the switch block at `palindrome`.



Since the current tapehead state is `b`, we execute the basic block at lines 21-22. So, we change the tapehead value to `blank`, move to the right. This is a while block, so the next block to be executed is still the switch block.



At this point, the tapehead index moves between the blank values as we move to the basic block at line 3. Then, the execution terminates and we accept the tape.

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

B.1 Complete TML Programs

When we defined execution of a valid TML program on a tape above, 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.

```
alphabet = {"a", "b"}
module simpleProgram {
    changeto b
    move left
    move right
    accept
}
```

After applying step 1 of completion, we get the following program.

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

After applying step 2, we have 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, after applying step 3, we 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 valid tape T in the same way. That is,*

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

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

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

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

- At the start, we have the same tape T for both P and P^+, with tapehead index 0. Moreover, the corresponding (completed) module of the first block in the first module of P is the first module of P. So, the result is true if $n = 0$.
- Now, assume that the result is true for some integer n, where the block b_n in the TML program P does not end with a terminating *flow* command. Let 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.

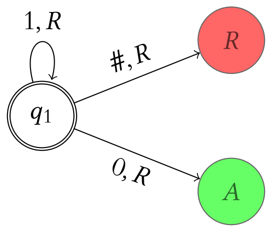
In that case, P and P^+ execute on T the same way by induction. □

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

B.2 Equivalence of TMs and TMLs

In this section, we will show that there is an equivalence between TMs and valid (complete) 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 complete TML program, and show the equivalence in this case as well.

We will first illustrate how to convert a TM to a (complete) TML program. So, consider the following TM: Consider the following TM:



Then, its corresponding TML program is the following:

```
alphabet = {"0", "1"}
module has0 {
   while 1 {
      changeto 1
      move right
   } 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 $\delta(q, \sigma) = (q', \sigma', \texttt{dir})$. We add a case in the *switch* command corresponding to letter σ (an *if* case if $q' \neq q$, otherwise a *while* case) with the following commands:
  - `changeto` σ'
  - `move` *dir*
  - in the case of an *if* block, 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 valid tape T in the same way. That is,*

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

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

- At the start, we have the same tape T for both M and P, with tapehead index 0. Moreover, the first module in P corresponds to the initial state q_0. So, the result is true if $n = 0$.
- Now, assume that the result is true for some integer n, where the TM state q_n is not `accept` or `reject`. In that case, $T_n = S_n$, $i_n = j_n$ and m_n is the corresponding module for q_n. Let 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 TM 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 (complete) TML program. This process is essentially the inverse of the one we saw converting a complete TML program to a TM. In particular, for each module m in P, we construct the state q as follows- for each letter 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 complete TM program:

```
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 the following:

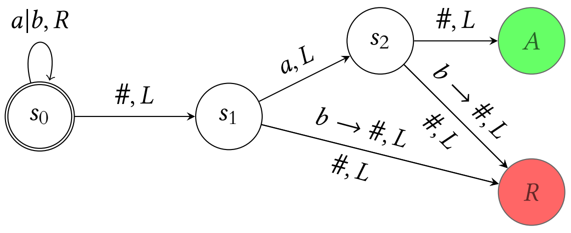


Figure B.1: *The TM corresponding to the program above. The state s_0 corresponds to the module* `moveToEnd`*; the state s_1 corresponds to the module* `checkAFirst`*; and the state s_2 corresponds to the module* `checkASecond`*.*

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

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

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

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

C Evaluation Content

C.1 Worksheet

C.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.

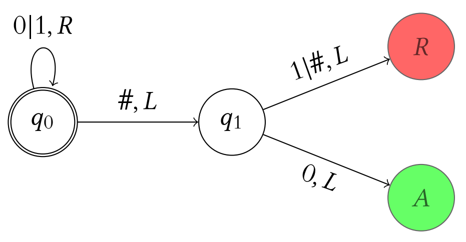
- `isDiv2`:

```
// 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`:

```
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
      }
   }
}
```

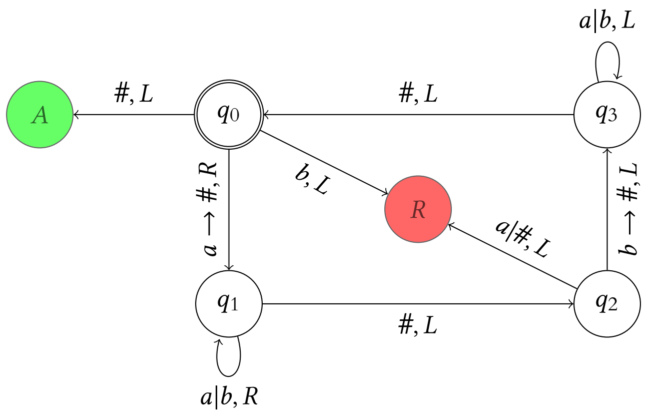
Both `isDiv2` and `isDiv2Rec` correspond to the following Turing Machine (TM):



- `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 following TM:



C.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
                  }
               }
            }
         }
      }
   }
}
```

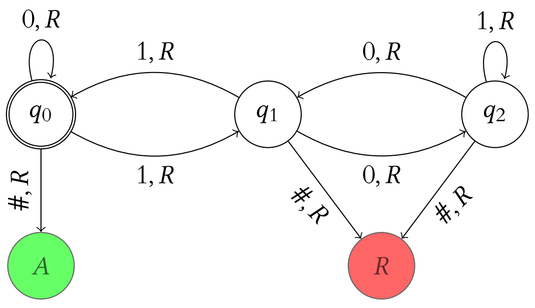
(a) Does the program accept the values:
   i. *ab*
   ii. *aabb*
   iii. *abba*
   iv. *bab*

(b) Describe the values this program accepts.

C.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!).

1. Consider the following TM FSM:

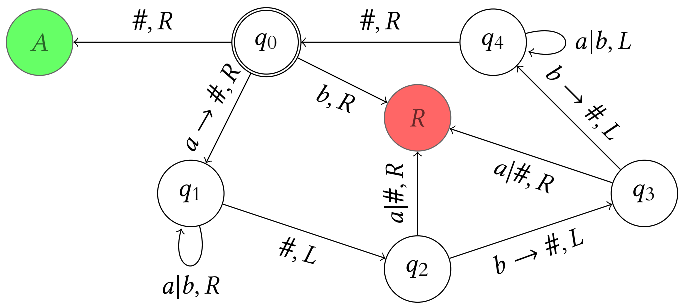


   You are given a basic representation of this FSM as code in Teams. The file is called mystery3.

   (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 following TM FSM:



   You are given a basic representation of this FSM 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.

C.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

C.2 Checklist

C.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.

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

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

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