



University  
of Glasgow | School of  
Computing Science

Honours Individual Project Dissertation

# TURING MACHINE LANGUAGE

**Pete Gautam**

November 30, 2022

## Abstract

## Acknowledgements

# Education Use Consent

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

Signature: Pete Gautam    Date: 27 September 2022

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>2</b>
2.1	Turing Machines	2
2.2	Parser	3
<b>3</b>	<b>Requirements</b>	<b>4</b>
<b>4</b>	<b>Design</b>	<b>5</b>
4.1	Turing Machine Language Program	5
4.2	Complete TML Programs	7
4.3	Equivalence of TMs and TMLs	8
<b>5</b>	<b>Implementation</b>	<b>10</b>
<b>6</b>	<b>Evaluation</b>	<b>11</b>
<b>7</b>	<b>Conclusion</b>	<b>12</b>
<b>A</b>	<b>TML EBNF</b>	<b>13</b>
	<b>Appendices</b>	<b>13</b>
<b>B</b>	<b>Incorrect TML Programs</b>	<b>14</b>
<b>C</b>	<b>TML Examples</b>	<b>16</b>
C.1	Executing a TML program on a Tape	16
C.2	Completing a TML program	18
C.3	Converting a TM to a complete TML program	20
	<b>Appendices</b>	<b>22</b>
<b>A</b>	<b>Proofs of the Theorems</b>	<b>22</b>
	<b>Bibliography</b>	<b>25</b>

# 1 | Introduction

## 2 | Background

In this chapter, we review the concept of Turing Machines and constructing a parser for a programming language.

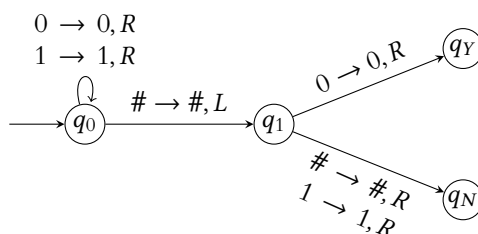
### 2.1 Turing Machines

In this section, we review the definition of Turing Machines and executing a TM on a valid tape.

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

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

We can represent a TM as a directed graph, with vertices as states and edges as transitions. For example, the following is a TM:



**Figure 2.1:** A Turing Machine that accepts binary numbers divisible by 2.

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  $q_Y$  and the reject state  $q_N$ . Every edge corresponds to an evaluation of the transition function  $\delta$ , e.g.  $\delta(q_0, a) = (q_1, \text{blank}, \text{right})$ .

Let  $\Sigma$  be an alphabet. A *tape*  $T$  on  $\Sigma$  is a function  $T: \mathbb{Z} \rightarrow \Sigma^+$ . 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 if  $x \geq a$  and  $x \leq b$ . We will represent a tape using a figure. For instance, let  $\Sigma = \{0, 1\}$ , and let  $T$  be the tape on  $\Sigma$  given below:

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

Then, the following figure represents the tape  $T$ :

$$\_ \quad \_ \quad \_ \quad \_ \quad \_ \quad \_$$

$\ominus \quad 1 \quad \ominus \quad \ominus$

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  $\Sigma$ , and let  $T$  be a (valid) tape on  $\Sigma$ . We execute  $M$  on  $T$  inductively, as follows:

- At any point during execution, we maintain 3 objects- a tape on  $\Sigma$ , a 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  $q_Y$  or  $q_N$ ). Denote  $\delta(q, t) = (q', t', \text{dir})$ . Then, the next state is  $q'$ , and the next tape  $S'$  and the next tapehead index  $j'$  are given by:

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

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

## 2.2 Parser

TODO



## 3 | Requirements

## 4 | Design

In this chapter, we define the Turing Machine Language (TML) as an alternative to Turing Machines (TM). We will define the syntax of a TML program and how it can be executed on a tape in a similar manner to TMs. We will then prove that there is an equivalence between TML programs and TMs in terms of their execution on a tape.

### 4.1 Turing Machine Language Program

In this section, we will define the syntax of the Turing Machine Language 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      switch tapehead {
4          while 0, 1 {
5              move right
6          } if blank {
7              move left
8              switch tapehead {
9                  if 0 {
10                     changeto blank
11                     accept
12                 } if 1, blank {
13                     changeto blank
14                     reject
15                 }
16             }
17         }
18     }
19 }
```

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* (*changeneto*, *move* or *flow* command). A basic block consists of at least one basic command, but it is not necessary for a basic block to be

composed of all the basic commands. If multiple commands are present in a basic block, they must be in the following order- *changeto*, *move* and *flow* command. In the program above, there are many basic blocks, e.g. at lines 5, 7, 10-11 and 13-14. We do not say that line 10 is a basic block by itself; we want the basic block to be as long as possible.

- A *switch block* consists of a single switch statement. A switch statement must contain precisely one case (*if* or *while* command) for each of the letter in the alphabet, including the blank letter. The first block of a case block must be a basic block. In the program above, there is a switch block at lines 8-16.
- 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 6-17; the *if* block has a basic block at line 7 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 any other block, basic or switch.

The EBNF for the TML and examples of syntax errors are given in the appendix.

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

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

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

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

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

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

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

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

\* otherwise, execution is terminated and we return the state `reject`.

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

An example that illustrates the execution process is given in the appendix.

## 4.2 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.

```

1  alphabet = {"0", "1"}
2  module isOdd {
3      // move to the end
4      switch tapehead {
5          while 0 {
6              changeto 0
7              move right
8          } while 1 {
9              changeto 1
10             move right
11         } if blank {
12             changeto blank
13             move left
14             goto isOddCheck
15         }
16     }
17 }
18 module isOddCheck {
19     // accept if and only if the value is 1
20     switch tapehead {
21         if 0, blank {
22             changeto 0
23             move left
24             reject
25         } if 1 {
26             changeto blank
27             move left
28             accept
29         }
30     }
31 }

```

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.

An example that illustrates this process is given in the appendix.

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

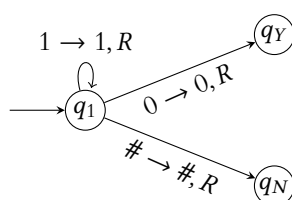
The proof of the theorem is given in the appendix.

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

### 4.3 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:

```

1 alphabet = {"0", "1"}
2 module has0 {
3   switch tapehead {
4     while 1 {
5       changeto 1
6       move right
7     } if 0 {
8       changeto 1

```

```

9      move right
10     accept
11   } if blank {
12     changeto blank
13     move left
14     reject
15   }
16 }
17 }

```

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

- the module contains a single *switch* command;
- for each letter  $\sigma$  in the alphabet  $\Sigma^+$ , denote  $\delta(q, \sigma) = (q', \sigma', \text{dir})$ . We add a case in the *switch* command corresponding to letter  $\sigma$  (an *if* case if  $q' \neq q$ , otherwise a *while* case) with the following commands:
  - *changeto*  $\sigma'$
  - *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  $\Sigma$ ;
- modules corresponding to every state  $q$  in  $M$ ;
- the module corresponding to the initial state  $q_0$  placed at the top.

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

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

The proof of the theorem is given in the appendix.

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

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

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

**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.*

The proof of the theorem is given in the appendix.

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

## 5 | Implementation

## 6 | Evaluation



## 7 | Conclusion

## A | TML EBNF

The following is the EBNF for the Turing Machine Language.

```

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

```

## B | Incorrect TML Programs

Below is a selection of incorrect TML programs, along with an explanation as to why they are incorrect.

- **Invalid while block (flow command)**

```

1  alphabet = {a, b}
2  module main {
3      switch tapehead {
4          while a, b {
5              accept
6          }
7      }
8  }
```

A while block cannot have a flow command. We know that the next block to be executed is the same block, so this doesn't make sense- are we meant to terminate or re-run the current block?

- **Invalid while block (multiple blocks)**

```

1  alphabet = {a, b}
2  module main {
3      switch tapehead {
4          while blank {
5              move left
6              move right
7          }
8      }
9  }
```

A while block must be composed of a single basic block. We know that a while block corresponds to a self-loop, so if we have 2 blocks, we would need another state to link to the current state. This doesn't make sense, so we can only have one block. The block must also be a simple block since we have already fixed what character the while block applies to.

- **Invalid switch block (first block not basic)**

```

1  alphabet = {a, b}
2  module main {
3      switch tapehead {
4          if a, blank {
5              accept
6          }
7      }
8  }
```

The first block in a switch block must be a basic block. A switch block corresponds to a state that branches to different states. So, the first block in a switch block must be a basic block so we know the transitions for the branch going into the new states.

- **Invalid module (flow command not final)**

```

1  alphabet = {a, b}
2  module main {
3      goto simple
4      move left
5  }
6  module simple {
7      reject
8  }
```

If we have a *flow*-command in a block, then it must be the final command. A *flow*-command moves to terminating execution or executing a module from the start, so any command below it cannot be executed. In this case, we are moving to a different block called `simple`, so we never move `left`.

- **Invalid module (switch block not final)**

```

1  alphabet = {a, b}
2  module a {
3      switch tapehead {
4          if a, b {
5              changeto blank
6              move right
7              move right
8              changeto b
9          } if blank {
10             move left
11             changeto a
12             reject
13         }
14     }
15     accept
16 }
```

If a module has a *switch* block, it must be the final block. In this case, that is not the case. It is possible that the program is meant to accept if we execute the *if*-block at line 4, but it is not possible to continue the *if*-block at line 9- this block already ends with a flow command. So, the execution would be unclear if this was allowed.

- **Invalid module (invalid name)**

```

1  alphabet = {a, b}
2  module accept {
3      move right
4  }
```

A module cannot be named `accept` or `reject`. It can be named any of the other keywords, but not `accept` or `reject`.

# C | TML Examples

## C.1 Executing a TML program on a Tape

Consider the following TML program:

```

1  alphabet = {"a", "b"}
2  module palindrome {
3      switch tapehead {
4          if blank {
5              accept
6          } if a {
7              changeto blank
8              move right
9              switch tapehead {
10                 while a, b {
11                     move right
12                 } if blank {
13                     move left
14                     switch tapehead {
15                         if blank, a {
16                             changeto blank
17                             move left
18                             goto restart
19                         } if b {
20                             reject
21                         }
22                     }
23                 }
24             }
25         } if b {
26             changeto blank
27             move right
28             switch tapehead {
29                 while a, b {
30                     move right
31                 } if blank {
32                     move left
33                     switch tapehead {
34                         if blank, b {
35                             changeto blank
36                             move left
37                             goto restart
38                         } if a {
39                             reject
40                         }
41                     }
42                 }

```

```

43     }
44   }
45 }
46 }
47 module restart {
48   switch tapehead {
49     while a, b {
50       move left
51     } if blank {
52       move right
53       goto palindrome
54     }
55   }
56 }

```

We will execute the program on the following tape.

$$\begin{array}{ccccccc} & a & b & a & & & \\ - & \frac{}{} & - & - & - & - & \\ & \uparrow & & & & & \end{array}$$

The arrow points to the tapehead value. We first execute the switch block at `palindrome`. Since the tapehead value is `a`, we execute the basic block at lines 7–8. 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 9–24. Now, the current tape is the following.

$$\begin{array}{ccccccc} & b & a & & & & \\ - & \frac{}{} & - & - & - & - & \\ & \uparrow & & & & & \end{array}$$

The current block is a switch block. The tapehead value is `b`, so we are at the *while* command at line 11. 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.

$$\begin{array}{ccccccc} & b & a & & & & \\ - & - & \frac{}{} & - & - & - & \\ & & \uparrow & & & & \end{array}$$

The current block is still a switch block. The tapehead value is `a`, so we execute the same *while* command at line 11. Moreover, the next block to execute is still the switch block. Now, the current tape state is the following.

$$\begin{array}{ccccccc} & b & a & & & & \\ - & - & - & \frac{}{} & - & - & \\ & & & \uparrow & & & \end{array}$$

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 13. 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 14–22.

$$\begin{array}{c} \text{b} \quad \text{a} \\ - \quad - \\ \uparrow \end{array}$$

Now, the current block is a switch block. The tapehead value is **a**, so we take the basic block at lines 16–18. The value of the tapehead becomes **blank**, and the tapehead moves to the left. The next block to execute is the switch block in **restart**.

$$\begin{array}{c} \text{b} \\ - \quad - \\ \uparrow \end{array}$$

Since the current tapehead value is **b**, we execute the basic block at line 50. 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.

$$\begin{array}{c} \text{b} \\ - \quad - \\ \uparrow \end{array}$$

Since the current tapehead state is **blank**, we execute the basic block at lines 52–53. So, we move to the right, and the tape is left as is. The next block to execute is the switch block at **palindrome**.

$$\begin{array}{c} \text{b} \\ - \quad - \\ \uparrow \end{array}$$

Since the current tapehead state is **b**, we execute the basic block at lines 26–27. 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.

$$\begin{array}{c} - \quad - \\ \uparrow \end{array}$$

At this point, the tapehead index moves between the blank values as we move to the basic block at line 5. Then, the execution terminates and we accept the tape.

## C.2 Completing a TML program

The steps to convert a TML program to complete it is the following:

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.

We illustrate this 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  }

```

After applying step 1 of completion, 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 }

```

After applying step 2, we have the following program.

```

1  alphabet = {"a", "b"}
2  module simple1 {
3      switch tapehead {
4          if a, b, blank {
5              changeto b
6              move left
7              goto simple2
8          }
9      }
10 }
11 module simple2 {
12     switch tapehead {
13         if a, b, blank {
14             move right
15             accept
16         }
17     }
18 }

```

Finally, after applying step 3, we get the following program:

```

1  alphabet = {"a", "b"}
2  module simple1 {
3      switch tapehead {
4          if a {
5              changeto b
6              move left
7              goto simple2

```



```

8      } if b {
9          changeto b
10         move left
11         goto simple2
12     } if blank {
13         changeto b
14         move left
15         goto simple2
16     }
17 }
18 }
19 module simple2 {
20     switch tapehead {
21         if a {
22             changeto a
23             move right
24             accept
25         } if b {
26             changeto b
27             move right
28             accept
29         } if blank {
30             move right
31             accept
32         }
33     }
34 }

```

This program obeys the definition of a complete program.

### C.3 Converting a TM to a complete TML program

Consider the following complete TM program:

```

1  alphabet = {"a", "b"}
2  module moveToEnd {
3      switch tapehead {
4          while a {
5              changeto a
6              move right
7          } while b {
8              changeto b
9              move right
10         } if blank {
11             changeto blank
12             move left
13             goto checkAFirst
14         }
15     }
16 }
17 module checkAFirst {
18     switch tapehead {
19         if a {
20             changeto blank

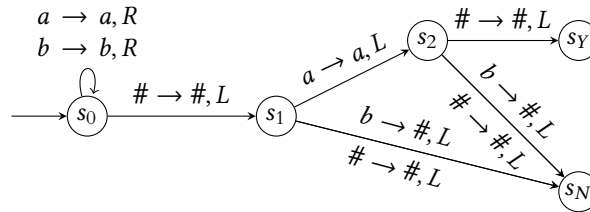
```

```

21     move left
22     goto checkASecond
23   } if b, blank {
24     changeto blank
25     move left
26     reject
27   }
28 }
29 }
30 module checkASecond {
31   switch tapehead {
32     if a {
33       changeto blank
34       move left
35       accept
36     } if b, blank {
37       changeto blank
38       move left
39       reject
40     }
41   }
42 }

```

Then, its corresponding TM is the following:



**Figure C.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`.

## A | Proofs of the Theorems

**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  $\sigma_n$  be the letter at index  $i_n = j_n$  on the tape  $S_n = T_n$ .
  - If the *changeto* command is missing in  $b_n$  for  $\sigma_n$ , then the next tape  $T_{n+1} = T_n$ . In the complete module  $m_n$ , the case for  $\sigma_n$  will have the command *changeto*  $\sigma_n$ . So, the next tape is given by:

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

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

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

- \* Otherwise, both  $P$  and  $P^+$  have the same flow command, meaning that there is either correspondence between the next module to be executed, or both the program terminate with the same status.

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

**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  $\sigma_n$  be the letter at index  $i_n = j_n$  on the tape  $T_n = S_n$ . Denote  $q(q_n, \sigma_n) = (q_{n+1}, \sigma_{n+1}, \text{dir})$ . In that case,

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

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

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

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

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

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

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

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

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

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

**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  $\sigma_n$  be the letter at index  $j_n = i_n$  on the tape  $S_n = T_n$ . We now consider the single switch block in  $m_n$ :

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

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

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

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

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

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

Now, we consider the flow command:

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

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

So, the result follows from induction. □

## Bibliography