University of Glasgow | School of Computing Science

Turing Machine Language

Pete Gautam
February 21, 2023

Abstract

Acknowledgements

Education Use Consent

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

Signature: Pete Gautam   Date: 27 September 2022

Contents

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, Σ, δ, q_0), where:

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

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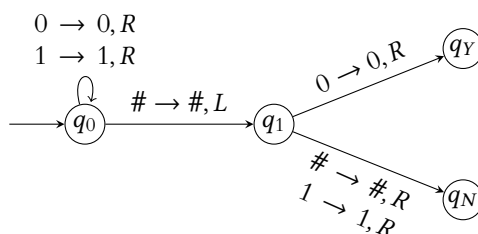

Figure 2.1: A Turing Machine that accepts binary numbers divisible by 2.

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 δ, e.g. $\delta(q_0, a) = (q_1, \texttt{blank}, \texttt{right})$.

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 will 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:

```
    0   1   0   0
 _  _   _   _   _   _
```

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 q_Y or q_N). Denote $\delta(q, t) = (q', t', \mathtt{dir})$. Then, the next state is q', and the next tape S' and the next tapehead index j' are given by:

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

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

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 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 | Turing Machine Language

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     while 0, 1 {
4        move right
5     } if blank {
6        move left
7        if 0 {
8           changeto blank
9           accept
10       } if 1, blank {
11          changeto blank
12          reject
13       }
14    }
15 }
```

A program in TML 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 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 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.

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       while 0 {
5           changeto 0
6           move right
7       } while 1 {
8           changeto 1
9           move right
10      } if blank {
11          changeto blank
12          move left
13          goto isOddCheck
14      }
15  }
16  module isOddCheck {
17      // accept if and only if the value is 1
18      if 0, blank {
19          changeto 0
20          move left
21          reject
22      } if 1 {
23          changeto blank
24          move left
25          accept
26      }
27  }
```

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

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

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

1. We first break each module into smaller modules so that every module has just one basic/switch block- we add a *goto* command to the next module if it appeared just below this block.

2. Then, we can convert each basic block to a switch block by just adding a single case that applies to each letter in the alphabet.

3. Finally, we add the default values to each basic block to get a complete TML program.

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

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:
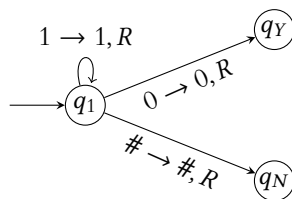
Then, its corresponding TML program is the following:

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

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:
  - $\texttt{changeto}\ \sigma'$
  - $\texttt{move}\ dir$
  - in the case of an *if* block, if q' is $\texttt{accept}$, then the command $\texttt{accept}$; if q' is $\texttt{reject}$, then the command $\texttt{reject}$; otherwise, $\texttt{goto}\ q'$.

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.*

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 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 $\texttt{dir}$ is the direction given in the *move* command within m;
- if the *flow* command in m corresponding to σ is $\texttt{accept}$, then q' is the $\texttt{accept}$ state; if it is $\texttt{reject}$, then q' is the $\texttt{reject}$ 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*. 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 | Turing Machine Parser and Website

5.1 Parser

5.1.1 Design

5.1.2 Implementation

5.2 Website

5.2.1 Design

5.2.2 Implementation

6 | Evaluation

7 | Conclusion

A | TML EBNF

The following is the EBNF for the Turing Machine Language.

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

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     while a, b {
4        accept
5     }
6  }
```

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     while blank {
4        move left
5        move right
6     }
7  }
```

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     if a, blank {
4        if a {
5           move right
6        }
7     }
8  }
```

The first block in a switch block must be a basic block. That is, we cannot have nested if blocks without an intermediate command. This does not make sense semantically- in the case above, we should just have a single if block for a.

- **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       if a, b {
4           changeto blank
5           move right
6           move right
7           changeto b
8       } if blank {
9           move left
10          changeto a
11          reject
12      }
13      accept
14  }
```

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 3, but it is not possible to continue the *if*-block at line 8- 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       if blank {
4           accept
5       } if a {
6           changeto blank
7           move right
8           while a, b {
9               move right
10          } if blank {
11              move left
12              if blank, a {
13                  changeto blank
14                  move left
15                  goto restart
16              } if b {
17                  reject
18              }
19          }
20      } if b {
21          changeto blank
22          move right
23          while a, b {
24              move right
25          } if blank {
26              move left
27              if blank, b {
28                  changeto blank
29                  move left
30                  goto restart
31              } if a {
32                  reject
33              }
34          }
35      }
36  }
37  module restart {
38      while a, b {
39          move left
40      } if blank {
41          move right
42          goto palindrome
```

```
43        }
44    }
```

We will execute the program on the following tape.

$$\_ \; \underset{\uparrow}{\underline{a}} \; \underline{b} \; \underline{a} \; \_$$

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.

$$\_ \; \underset{\uparrow}{\underline{b}} \; \underline{a} \; \_$$

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.

$$\_ \; \underline{b} \; \underset{\uparrow}{\underline{a}} \; \_$$

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.

$$\_ \; \underline{b} \; \underline{a} \; \underset{\uparrow}{\_}$$

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.

$$\_ \; \underline{b} \; \underset{\uparrow}{\underline{a}} \; \_$$

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`.

$$\_ \; \underset{\uparrow}{\underline{b}} \; \_$$

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.

$$\underset{\uparrow}{\_} \ \overset{b}{\_} \ \_$$

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`.

$$\_ \ \underset{\uparrow}{\overset{b}{\_}} \ \_$$

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.

$$\_ \ \_ \ \underset{\uparrow}{\_}$$

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.

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       if a, b, blank {
4           changeto b
5           move left
6           goto simple2
7       }
8    }
9    module simple2 {
10      if a, b, blank {
11          move right
12          accept
13      }
14   }
```

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

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

```
28          accept
29      }
30  }
```

This program obeys the definition of a complete program.

C.3 Converting a TM to a complete TML program

Consider the following complete TM program:

```
1   alphabet = {"a", "b"}
2   module moveToEnd {
3       while a {
4           changeto a
5           move right
6       } while b {
7           changeto b
8           move right
9       } if blank {
10          changeto blank
11          move left
12          goto checkAFirst
13      }
14  }
15  module checkAFirst {
16      if a {
17          changeto blank
18          move left
19          goto checkASecond
20      } if b, blank {
21          changeto blank
22          move left
23          reject
24      }
25  }
26  module checkASecond {
27      if a {
28          changeto blank
29          move left
30          accept
31      } if b, blank {
32          changeto blank
33          move left
34          reject
35      }
36  }
```
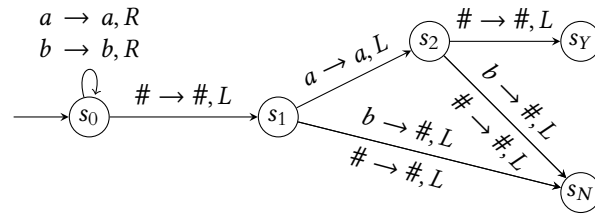
Then, its corresponding TM is the following:

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 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. □

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. □

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. $\qquad\square$

Bibliography