TML Specification

Pete Gautam

October 4, 2022

Abstract

This document contains the specification of the TM programming language (called Turing Machine Language, or TML), and how it can be used to change the state of a tape. In the first section, we discuss TML rules (the EBNF and contextual rules), and give examples of valid and invalid TML programs. In the second section, we define tapes and how a TML program can be executed on a tape.

1 TML rules

In this section, we define the rules that a valid program in TML should obey. First, the following is the specification of the TML in EBNF:

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

Next, we analyse the rules of a TML program.

Rule 1.1. A valid TML program is composed of the *alphabet*, followed by one or more *modules*.

Rule 1.2. A module contains a collection of a *blocks* (a specific sequence of commands). There are two types of blocks- *basic blocks* and *switch blocks*.

Rule 1.3. 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.

Example 1.4. The following is a simple TML program:

```
1 alphabet = {"a", "b"}
2 module simple {
3     changeto blank
4     move right
5     changeto a
6     move left
7     accept
8 }
```

It is composed of a single module, called `simple`. The module has 2 basic blocks- lines 3-4, and lines 5-7.

Remark 1.5. In the example above, we could have also said that there were 5 basic blocks, one in each line from line 3 to line 7. If it is possible for us to break a module into blocks in different ways, we always choose the one that ends up with the fewest number of blocks. In this case, we cannot have just one block since lines 3-5 do not form a basic block- there are two *changeto* commands. So, we have 2 basic blocks above.

Rule 1.6. 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.

Rule 1.7. The body of an *if* command can be composed of multiple blocks. These blocks can be both basic blocks and switch blocks.

Rule 1.8. The body of a *while* command must be composed of a single basic block. The basic block cannot have a *flow* command. Further, a switch block must be the final block present; it cannot be followed by any other block, basic or switch.

Example 1.9. The following is another 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                      accept
11                  } if 1, blank {
12                      reject
13                  }
14              }
15          }
16      }
17  }
```

It is composed of a single module `isEven`. The module has:

- a switch block at lines 3-16;
- a basic block at line 5;
- a basic block at line 7;
- a nested switch block at lines 8-14;
- a basic block at line 10; and
- a basic block at line 12.

2 TML execution

In this section, we discuss how a valid TML program can execute a tape. The tape we will be using has infinite spaces in both directions.

Definition 2.1. Let Σ be an alphabet. A *tape* T *on* Σ is a function $T : \mathbb{Z} \to \Sigma^+$, where $\Sigma^+ = \Sigma \cup \{\texttt{blank}\}$.

Remark 2.2. A valid TML program always specifies the alphabet. This corresponds to the alphabet Σ used in the tape.

Although there are many possible tapes, we will only be interested in tapes that obey some rules.

Definition 2.3. Let Σ be an alphabet and let T be a tape on Σ. Then, 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 $x \leq b$.

Remark 2.4. In a valid tape, there are only finitely many values that are non-`blank` and there is no gap between any two non-`blank` values.

Remark 2.5. We require the tape to have finitely many non-`blank` entries so that we start execution with the tapehead at the first non-`blank` entry. Moreover, this ensures that going from the start of the tape to the end does not create

an infinite loop- this is a very common procedure. If we allowed for infinitely many non-`blank` entries, we would need to specify where the initial location of the tapehead is. Moreover, there would be much fewer programs that we could write which are guaranteed to terminate.

Remark 2.6. A valid tape can always be represented as an illustration. For instance, consider the following valid tape T on $\{0, 1\}$:

$$T(x) = \begin{cases} 0 & x \in \{1, 3, 4\} \\ 1 & x \in \{2\} \\ \texttt{blank} & \text{otherwise.} \end{cases}$$

Then, an illustration of the tape is:

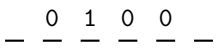


Figure 1: The tape as a diagram.

Only the non-`blank` values are represented. Blank ones are represented by empty lines.

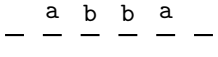
Remark 2.7. More than one tape definition can result in the same diagram. In the case above, the following tape S on $\{0, 1\}$ also has the same diagram.

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

Example 2.8. Consider the following TML program:

```
alphabet = {"a", "b"}
module isSecondValueA {
    move right
    switch tapehead {
        if a {
            accept
        } if b, blank {
            reject
        }
    }
}
```

The following is a valid tape for the program:



Now, we will define how we can execute a TML program on a valid tape.

Definition 2.9. Let P be a TML program, let T be a valid tape for the program with i the smallest integer such that $T(i)$ is not `blank` (i is the original *tapehead index* and $T(i)$ the original *tapehead value*). We *execute* P *on* T by constructing (countable) different tapes until execution is terminated. We first take the given tape and the tapehead index and execute it using the first block b in the first module of P to construct the next tape T' and next tapehead index i'. This is done as follows:

- if the block is a switch block, we take the first block from the case corresponding to the tapehead value- now, we must have a basic block;
- if there is a *changeto* `val` command in the basic block, 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 there is a *move* `dir` command in the basic block, 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 $b' = b$, i.e. we execute this switch block again (not necessarily the same case block);
  - if the block contains a terminating *flow* command, we terminate and return the terminated state;
  - 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 module, then b' is next block in this module;
  - otherwise, we terminate and return the state `reject`.

If execution is not terminated, we execute the block b' with the new tape T' and a new tapehead index i'.

Remark 2.10. By construction, for a valid program, precisely one of the 5 possible step applies when choosing the next block to execute or terminate.