TML SPECIFICATION

PETE GAUTAM

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. This section also includes some examples of executing a TM program on a given 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:

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

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*.

1

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:

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

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:

```
alphabet = {"0", "1"}
module isEven {
    switch tapehead {
        while 0, 1 {
            move right
        } if blank {
            move left
            switch tapehead {
                if 0 {
                    accept
```

```
            } if 1, blank {
                reject
            }
        }
    }
  }
}
```

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 $\{0, 1\}$:

$$T(x) = \begin{cases} 0 & x \in \{1, 3, 4\} \\ 1 & x \in \{2\} \\ \texttt{blank} & \text{otherwise.} \end{cases}$$
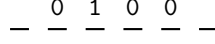
Then, an illustration of the tape is:

```
  0  1  0  0
__ __ __ __ __
```

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.

$$S(x) = \begin{cases} 0 & x \in \{0, 2, 3\} \\ 1 & x \in \{1\} \\ \texttt{blank} & \text{otherwise.} \end{cases}$$
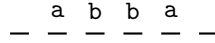
Example 2.8. Consider the following TML program:

```
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:

```
  a  b  b  a
__ __ __ __ __
```

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 initial *tapehead index* and $T(i)$ the initial *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.

Example 2.11. Consider the following TML program:

```
alphabet = {"a", "b"}
module palindrome {
   switch tapehead {
      if blank {
          accept
      } if a {
          changeto blank
          move right
          switch tapehead {
             while a, b {
                 move right
             } if blank {
                 move left
                 switch tapehead {
                    if blank, a {
                        changeto blank
                        move left
                        goto restart
                    } if b {
                        reject
                    }
                 }
             }
          }
      } if b {
```

```
            changeto blank
            move right
            switch tapehead {
                while a, b {
                    move right
                } if blank {
                    move left
                    switch tapehead {
                        if blank, b {
                            changeto blank
                            move left
                            goto restart
                        } if a {
                            reject
                        }
                    }
                }
            }
        }
    }
}
module restart {
    switch tapehead {
        while a, b {
            move left
        } if blank {
            move right
            goto palindrome
        }
    }
}
```

We will execute the program on the following tape.

$$- \quad \underset{\uparrow}{\text{a}} \quad \text{b} \quad \text{a} \quad -$$

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.

$$- \quad \underset{\uparrow}{\text{b}} \quad \text{a} \quad -$$

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.

$$- \quad \underline{b} \quad \underset{\uparrow}{\underline{a}} \quad -$$

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.

$$- \quad \underline{b} \quad \underline{a} \quad \underset{\uparrow}{-}$$

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 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.

$$- \quad \underline{b} \quad \underset{\uparrow}{\underline{a}} \quad -$$

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`.

$$- \quad \underset{\uparrow}{\underline{b}} \quad -$$

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.

$$\underset{\uparrow}{-} \quad \underline{b} \quad -$$

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`.

$$- \quad \underset{\uparrow}{\underline{b}} \quad -$$

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.

$$\overline{} \;\; \overline{} \;\; \overline{} \atop \uparrow$$

The current tapehead state is blank, so we execute the basic block at line 32. We move to the left and keep the tape as is. The next block to execute is the switch block at lines 33-41.

$$\overline{} \;\; \overline{} \;\; \overline{}$$
$$\uparrow$$

The current tapehead value is blank, so we execute the basic block at lines 35-37. We keep the tape as is, and move to the left. The next block to execute is the switch block at lines 48-55.

$$\overline{} \;\; \overline{} \;\; \overline{}$$
$$\uparrow$$

The current tapehead value is blank, so we execute the basic block at lines 52-53. In that case, we move to the right and go to the switch block at palindrome.

$$\overline{} \;\; \overline{} \;\; \overline{}$$
$$\uparrow$$

The current tapehead value is blank, so we execute the basic block at line 5. We move to the left, keep the tapehead as is, and accept the tape. In that case, the final state of the tape is the following.

$$\overline{} \;\; \overline{} \;\; \overline{}$$
$$\uparrow$$