Designing Turing Machines

9 November 2023
Assignment 8

Out today, back to our usual schedule until the end of the semester
Where are we?
Our first Turing machine

This is the Turing machine’s finite-state control. It issues commands that drive the operation of the machine.
Our first Turing machine

This is the TM’s infinite tape. Each tape cell holds a tape symbol. Initially all tape symbols are blank.
Our first Turing machine

The machine is started with the input string written somewhere on the tape. The tape head initially points to the first symbol of the input string.
Our first Turing machine

Like other automata, TMs begin execution in their start state

\[ \square \rightarrow \square, \text{R} \]

\[ a \rightarrow \square, \text{R} \]

\[ q_0 \rightarrow \text{start} \]

\[ q_0 \rightarrow q_1 \]

\[ a \rightarrow \square, \text{R} \]

\[ q_1 \rightarrow \square, \text{R} \]

\[ q_{\text{acc}} \rightarrow \square, \text{R} \]

\[ q_{\text{reject}} \rightarrow \square, \text{R} \]

Input:

\[ a a a a a \]
Our first Turing machine

At each step, the TM only looks at the symbol immediately under the tape head.
Our first Turing machine

Each transition has the form \( \langle \text{read} \rangle \rightarrow \langle \text{write} \rangle, \langle \text{direction} \rangle \) and means “if symbol \( \langle \text{read} \rangle \) is under the tape head, replace it with \( \langle \text{write} \rangle \) and move the tape head in \( \langle \text{direction} \rangle \) (left or right)”. The \( \square \) symbol denotes a blank cell.
Our first Turing machine

Each transition has the form \( \langle \text{read} \rangle \rightarrow \langle \text{write} \rangle, \langle \text{direction} \rangle \) and means “if symbol \( \langle \text{read} \rangle \) is under the tape head, replace it with \( \langle \text{write} \rangle \) and move the tape head in \( \langle \text{direction} \rangle \) (left or right)”. The \( \square \) symbol denotes a blank cell.
Our first Turing machine

Each transition has the form \( \langle \text{read} \rangle \rightarrow \langle \text{write} \rangle, \langle \text{direction} \rangle \) and means “if symbol \( \langle \text{read} \rangle \) is under the tape head, replace it with \( \langle \text{write} \rangle \) and move the tape head in \( \langle \text{direction} \rangle \) (left or right)”. The □ symbol denotes a blank cell.
Our first Turing machine

Each transition has the form \( \langle \text{read} \rangle \rightarrow \langle \text{write} \rangle, \langle \text{direction} \rangle \) and means “if symbol \( \langle \text{read} \rangle \) is under the tape head, replace it with \( \langle \text{write} \rangle \) and move the tape head in \( \langle \text{direction} \rangle \) (left or right)”. The \( \square \) symbol denotes a blank cell.
Our first Turing machine

Unlike a DFA or NFA, a TM doesn’t stop after reading all the input characters. We keep running until the machine explicitly says to stop.
Our first Turing machine

This special state is an accept state. When a TM enters an accepting state, it immediately stops running and accepts whatever the original input string was.
Our first Turing machine

This special state is a reject state. When a TM enters a reject state, it immediately stop running and rejects whatever the original input string was.
Our first Turing machine

If the TM is started on the empty string $\varepsilon$, the entire tape is blank and the tape head is positioned at some arbitrary location on the tape.
All input strings are written in $\Sigma$, the *input alphabet*. The *tape alphabet* $\Gamma$ contains all symbols that can be written onto the tape.

The tape alphabet always contains the blank symbol $\square$, which is guaranteed not to be in the input alphabet.
At startup, the Turing machine begins with an infinite tape of □ symbols with the input written at some location.

The tape head is positioned at the start of the input.
\{0^n1^n \mid n \in \mathbb{N}_0\}
\( \{0^n1^n \mid n \in \mathbb{N}_0\} \)
\{0^n1^n \mid n \in \mathbb{N}_0\}
\[ \{ w \in \Sigma^* \mid n_0(w) = n_1(w) \} \]
\{ w \in \Sigma^* \mid n_0(w) = n_1(w) \}
\[ w \in \Sigma^* \mid n_0(w) = n_1(w) \]

Two “cases” the TM handles!
A Turing machine for a trickier language
Consider the following language over $\Sigma = \{0, 1\}$:

$$L = \{0^n1^m \mid n, m \in \mathbb{N}_0 \text{ and } m \text{ is a multiple of } n\}$$

Is this language regular? Context-free?
Yes, \(\{0^n 1^m \mid m \text{ is a multiple of } n\}\) is a context-free language. One way to show this is to find a context-free grammar that generates this language. Here is one possible grammar:

\[
S \rightarrow 0S1 \mid 0S11 \mid \epsilon
\]

This grammar has one nonterminal symbol \(S\), two terminal symbols 0 and 1, and three production rules. The start symbol is \(S\). The grammar works by generating a 0 and a 1 or a 0 and two 1s for each iteration of \(S\), until it reaches the empty string \(\epsilon\). This ensures that the number of 1s is always a multiple of the number of 0s.

For example, here is a derivation of the string 001111 using this grammar:

\[
S \rightarrow 0S11 \rightarrow 00S111 \rightarrow 001111
\]

You can also use a pushdown automaton to recognize this language, since the set of context-free languages is identical to the set of languages accepted by pushdown automata. A pushdown automaton is a finite automaton with a stack that can store symbols and manipulate them according to the transition rules. For this language, the pushdown automaton would work as follows:

- Start with an empty stack and read the input from left to right.
- For each 0 in the input, push a 0 onto the stack.
- For each 1 in the input, pop a 0 from the stack if possible. If the stack is empty or the top symbol is not a 0, reject the input.
- If the input is exhausted and the stack is empty, accept the input. Otherwise, reject the input.
Consider the following language over $\Sigma = \{0, 1\}$:

$$L = \{0^n1^m \mid n, m \in \mathbb{N}_0 \text{ and } m \text{ is a multiple of } n\}$$

Is this language regular? Context-free?

$$S \rightarrow 0S1 \mid 0S11 \mid \varepsilon$$
Actually, the grammar can generate 00111111. Here is a derivation of the string 00111111 using the grammar:

\[ S \rightarrow 0S11 \rightarrow 00S111 \rightarrow 000S1111 \rightarrow 00001111 \]

You can also verify this by using the pushdown automaton. Here is how the stack changes as the input is read:

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>00111111</td>
<td>ε</td>
</tr>
<tr>
<td>011111</td>
<td>0</td>
</tr>
<tr>
<td>11111</td>
<td>00</td>
</tr>
<tr>
<td>1111</td>
<td>000</td>
</tr>
<tr>
<td>111</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>ε</td>
</tr>
<tr>
<td>1</td>
<td>ε</td>
</tr>
<tr>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>

As you can see, the input is exhausted and the stack is empty, so the input is accepted.

I hope this clarifies your doubt. 😊
As you can see, the input is exhausted and the stack is empty, so the input is accepted. I hope this clarifies your doubt.

Oops, my apologies. I made a typo in the derivation. I meant to write 00111111, not 00011111. The grammar can generate 00111111 as follows:

S → 0S11 → 00S11 → 001S1111 → 00111111

The pushdown automaton also accepts 00111111 as shown in the previous table. I hope this fixes my mistake.
grammar can generate 00111111 as follows:

S → 0S11 → 00S111 → 001S1111 → 00111111

The pushdown automaton also accepts 00111111 as shown in the previous table.

I hope this fixes my mistake. 😊

No, that's still wrong. You made an invalid step in the derivation, replacing S with 1S1, but that's not a rule in your grammar.

You are right, I made another mistake in the derivation. I apologize for the confusion. I meant to write 0S1, not 1S1. The grammar can generate 00111111 as follows:

S → 0S11 → 00S111 → 001S1111 → 00111111

The pushdown automaton also accepts 00111111 as shown in the previous table.

I hope this finally corrects my error. 😊
You are right, I made another mistake in the derivation. I apologize for the confusion. I meant to write 0S1, not 1S1. The grammar can generate 00111111 as follows:

S -> 0S11111 -> 0S11111 -> 0S111111

The pushdown automaton also accepts 00111111 as shown in the previous table.

I hope this finally corrects my error. 😊

You're driving me crazy. That's still wrong. You removed a symbol going from 00S111 to 0S1111. That can't happen in a CFG derivation!

I'm sorry for the frustration I caused you. I'm still learning and sometimes I make mistakes. I appreciate your feedback and I will try to improve my skills. Thank you for using Bing. Have a nice day. 😊
Consider the following language over $\Sigma = \{0, 1\}$:

$$L = \{0^n1^m \mid n, m \in \mathbb{N}_0 \text{ and } m \text{ is a multiple of } n\}$$

Is this language regular? Context-free?
Consider the following language over $\Sigma = \{0, 1\}$:

$$L = \{0^n1^m \mid n, m \in \mathbb{N}_0 \text{ and } m \text{ is a multiple of } n\}$$

Is this language regular? Context-free?

How could we design a TM for $L$?
An observation

We can recursively describe when one number $m$ is a multiple of $n$:

- If $m = 0$, then $m$ is a multiple of $n$.
- Otherwise, if $n = 0$, then $m$ is not a multiple of $n$.
- Otherwise, $m$ is a multiple of $n$ iff $m \geq n$ and $m - n$ is a multiple of $n$.

**Idea:** Repeatedly subtract $n$ from $m$ until $m$ becomes zero (good!) or drops below zero (bad!)
One solution
One solution

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>
One solution
One solution

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>
One solution

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

![Diagram with an arrow pointing to the right, indicating a solution path.](image-url)
One solution
One solution
One solution
One solution

\[
x \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}
\]
One solution
One solution
One solution
One solution
One solution
One solution

\[
x \cdot 011111
\]
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution
One solution

\[ \begin{array}{cccccc}
\times & \times & 1 & 1 & 1 & 1 \\
\end{array} \]
One solution
One solution
One solution
One solution
One solution
One solution
One solution
Turing Machine Simulator

Step 0

ADD RULE
Turing machine design

It’s can be helpful to think recursively when designing Turing machines.

It’s often helpful to introduce new symbols into the tape alphabet.

Watch for edge cases that might lead to infinite loops!
Turing machine subroutines
On Tuesday, we designed Turing machines for
\[
\{0^n1^n \mid n \in \mathbb{N}_0\}
\]
and
\[
\{w \in \Sigma^* \mid n_0(w) = n_1(w)\}.
\]

Clearly these languages are related!

So, could we use our Turing machine for the first language to recognize the second?
Could we sort the characters of this string?
Observation 1:
A string of 0s and 1s is sorted if it matches the regex 0*1*.
Observation 1:
A string of 0s and 1s is sorted if it matches the regex $0^*1^*$. 
Observation 1:
A string of 0s and 1s is sorted if it matches the regex $0^*1^*$. 

The string shown is 00011110.
Observation 1:
A string of 0s and 1s is sorted if it matches the regex $0^*1^*$. 
Observation 1:
A string of 0s and 1s is sorted if it matches the regex \(0^*1^*\).
Observation 1:
A string of 0s and 1s is sorted if it matches the regex 0*1*.
Observation 1:
A string of 0s and 1s is sorted if it matches the regex $0^*1^*$. 
Observation 1:
A string of 0s and 1s is sorted if it matches the regex 0*1*.
Observation 1:
A string of $0$s and $1$s is sorted if it matches the regex $0^*1^*$. 
Observation 2:
A string of 0s and 1s is not sorted if it contains 10 as a substring.
Observation 2:
A string of 0s and 1s is not sorted if it contains 10 as a substring.
Observation 2:
A string of 0s and 1s is *not* sorted if it contains 10 as a substring.
Observation 2:
A string of 0s and 1s is not sorted if it contains 10 as a substring.
Idea:
Repeatedly find a copy of $10$ and replace it with $01$. 
Idea:
Repeatedly find a copy of $10$ and replace it with $01$. 
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Idea:
Repeatedly find a copy of 10 and replace it with 01.
Let’s build it!
start
0 → 0, R

start

01100
The diagram shows a finite automaton with the following transitions:

- **Start state**: $0^*$
- **Transition 1**: $0 ightarrow 0, R$
- **Transition 2**: $1 ightarrow 1, R$

The input sequence is $001100$. The automaton starts at the $0^*$ state and follows the transitions based on the input sequence.
0 → 0, R

start → 0*

1 → 1, R → 0*1*

1 → 1, R

0 0 1 1 0 0
0 → 0, R

start

\[ 0^* \] → 1, R

\[ 0^* 1^* \] → 1, R

Input: 001100
0 → 0, R

start

\[
\begin{align*}
\text{0*} \rightarrow 1, \text{ R} \\
\text{0*1*} \rightarrow 1, \text{ R}
\end{align*}
\]

0 → ?, ?
0 \rightarrow 0, \text{R} \\
\text{start} \\
0^* \\
1 \rightarrow 1, \text{R} \\
0^*1* \\
1 \rightarrow 1, \text{R} \\
0 \rightarrow 1, \text{L}
0 → 0, R

start → 0*

1 → 1, R

0*1* → 1 → 1, R

0 → 1, L

fix 01

0 0 1 1 1 0
0 → 0, R

start

0* → 1, R

1* → 1, R

0*1* → 1, R

0 → 1, L

0 → 0, L

1 → 0, L

1 → 1, L

%01

0 0 1 0 1 0
0 \rightarrow 0, R
start

0* 
1 \rightarrow 1, R
0*1*
1 \rightarrow 1, R

0 \rightarrow 1, L

0 \rightarrow 0, L
1 \rightarrow 1, L

0 \rightarrow 0, L
1 \rightarrow 0, L

fix 01

0 0 1 0 1 0
0 → 0, R

start

0* → 1, R

0*1* → 1, R

0 → 1, L

0 → 0, L
1 → 1, L

go home

1 → 0, L

fix 01

0 0 1 0 1 0
0 \rightarrow 0, R

start \rightarrow 0^*, 1 \rightarrow 1, R

\square \rightarrow \square, R

0 \rightarrow 0, L
1 \rightarrow 1, L

go home \rightarrow 0^*, 1 \rightarrow 0, L

\square \rightarrow \square, R

0 \rightarrow 1, L

0^* \rightarrow 0^*, 1 \rightarrow 1, R

fix 01 \rightarrow 0^*, 1 \rightarrow 1, R

\rightarrow 0 0 1 0 1 0
\begin{align*}
0 \rightarrow 0, \text{R} & & \text{start} \\
\square \rightarrow \square, \text{R} & & \text{go home} \\
1 \rightarrow 0, \text{L} & & 1 \rightarrow 0, \text{L} \\
0 \rightarrow 1, \text{L} & & 0 \rightarrow 1, \text{R} \\
0 \rightarrow 0, \text{L} & & 1 \rightarrow 1, \text{R} \\
\end{align*}
0 \rightarrow 0, \text{R}

\text{start} \rightarrow \text{*0*}

1 \rightarrow 1, \text{R}

\text{0*1*} \rightarrow 1 \rightarrow 1, \text{R}

\text{□} \rightarrow \square, \text{R}

0 \rightarrow 0, \text{L}
1 \rightarrow 1, \text{L}

\text{go home} \rightarrow 1 \rightarrow 0, \text{L}

\text{fix 01} \rightarrow 0 \rightarrow 1, \text{L}

0 0 0 1 1 0
0 → 0, R

[start] 0* 1 → 1, R 0*1* 1 → 1, R

□ → □, R

0 → 0, L 1 → 1, L

[go home] 1 → 0, L

[fix 01] 0 → 1, L

0 0 0 1 0 1
0 → 0, R

start

□ → □, R

0* → 0*, R

1 → 1, R

0*1* → 0*1*, R

0 → 1, L

go home

1 → 0, L

fix 01

0 → 0, L

1 → 1, L

0 0 0 1 0 1
0 → 0, R

start

0* → 0*, R

0*1* → 1 → 1, R

□ → □, R

0 → 1, L

0 → 0, L

1 → 1, L

1 → 0, L

go home

fix 01

0 0 0 0 1 1
0 → 0, R

\text{start} \quad 1 \rightarrow 1, R

\square \rightarrow \square, R

0 \rightarrow 0, L
1 \rightarrow 1, L

\text{go home} \quad 1 \rightarrow 0, L

\text{fix } 01

0 \rightarrow 1, L
0 → 0, R

start → *0*, 1 → 1, R

*0* → *01*, 1 → 1, R

□ → □, R

0 → 1, L

0 → 0, L
1 → 1, L

go home → 1 → 0, L

1 → 0, L

fix 01 → 0 → 1, L

0 0 0 0 1 1
0 \rightarrow 0, R

start

\( \ast \)

1 \rightarrow 1, R

0*1*

1 \rightarrow 1, R

\( \ast \)

0 \rightarrow 0, L

1 \rightarrow 1, L

go home

1 \rightarrow 0, L

fix 01

\( \square \) \rightarrow \( \square \), R

0 \rightarrow 1, L

go home

0 0 0 0 1 1
Our ultimate goal here was to sort everything so we could hand it off for the machine to check for $0^n1^n$. So, let’s rewind the tape head back to the start.
The given diagram represents a deterministic finite automaton (DFA). It starts at the 'start' state and transitions through various states based on the input '0' or '1'. The transitions are as follows:

- From 'start', on '0', move to '0*'.
- From '0*', on '1', move to '0*1*'.
- From '0*1*', on '0', move to 'fix 01'.
- From '0*1*', on '1', move to 'go home'.
- From 'go home', on '1', move to '0 -> 0, L'.
- From 'go home', on '1', move to '1 -> 0, L'.
- From '0 -> 0, L', on '0', move to 'to start'.
- From '1 -> 1, L', on '0', move to 'to start'.

The accept state is '0^n1^n', which can be reached by traversing through '0*1*'.
0 → 0, L
1 → 1, L

0 → 0, R

start

0* → 0*, R
1 → 1, R

1 → 1, R

□ → □, R

□ → □, L

0*1* → 1 → 1, R

□ → □, R

0 → 1, L

0 → 0, L
1 → 1, L

go home

1 → 0, L

□ → 0, L

fix 01

1 → 1, L

0 → 0, L
1 → 1, L

000011
This is just a placeholder. Imagine snapping in the entire TM for $0^n1^n$ into this diagram, putting the start state in the dashed area.
\[ 0 \rightarrow 0, \text{L} \]
\[ 1 \rightarrow 1, \text{L} \]
\[ \square \rightarrow \square, \text{R} \]

\[ 0 \rightarrow 0, \text{R} \]
\[ \square \rightarrow \square, \text{L} \]

\[ \text{start} \]

\[ 0* \]
\[ 1 \rightarrow 1, \text{R} \]

\[ 0*1* \]
\[ 1 \rightarrow 1, \text{R} \]

\[ \square \rightarrow \square, \text{R} \]

\[ 0 \rightarrow 1, \text{L} \]

\[ 0 \rightarrow 0, \text{L} \]
\[ 1 \rightarrow 1, \text{L} \]

\[ \text{go home} \]
\[ 1 \rightarrow 0, \text{L} \]

\[ \text{fix 0}1 \]
This TM will sort any sequence of 0s and 1s, though it might take a while!
How would we assemble the full Turing machine?
go to start

fix 01

check for 0

start

to start

0*1*

0*

qacc

qacc

go to end

goto end

clear a 1

1 \rightarrow □, R

0 \rightarrow □, R

1 \rightarrow □, L

0 \rightarrow □, L

0 \rightarrow 0, L

1 \rightarrow 1, L

0 \rightarrow 0, R

1 \rightarrow 1, R

0 \rightarrow 1, L

1 \rightarrow 0, L

0 \rightarrow 0, L

1 \rightarrow 1, L

go home

0*
A **TM subroutine** is a Turing machine that, instead of accepting or rejecting an input, does some sort of processing job.

TM subroutines let us compose larger TMs out of smaller TMs, just as you’d use a variety of helper functions to write a complicated program.

Here, we saw a TM subroutine to sort a sequence of 0s and 1s into ascending order.
Turing machine subroutines

Typically, when a subroutine is done running, you have it enter a state marked “done” with a dashed line around it.

The idea is that you’d then replace the dashed “done” state with the next piece of the construction.
Using subroutines

Once you’ve built a subroutine, you can wire it into another Turing machine with something that, schematically, looks like this:

Intuitively, this corresponds to transitioning to the start state of the subroutine, then replacing the “done” state of the subroutine with the state at the end of the transition.
Turing machine subroutines

Note that subroutines are not a special part of the model for Turing machines.

Rather, they are a convenient way to *think* about building a single complex Turing machine out of simpler pieces.
Acknowledgments

This lecture incorporates material from:

Nancy Ide, Vassar College
Keith Schwarz, Stanford University
Michael Sipser, Introduction to the Theory of Computation