SE 3310b

Theoretical Foundations of Software Engineering

Turing Machines

Aleksander Essex

Turing Machines

Introduction

We've finally arrived at a complete model of computation: Turing machines. Turing machines capture a model of computation that has:

- Full effective range of computational power: if you can compute it, there exists a TM that can compute it
- Fundamental limitations: Even a TM can't solve some problems

Western Engineering

Turing Machine Components

A Turing machine consists of the following components:

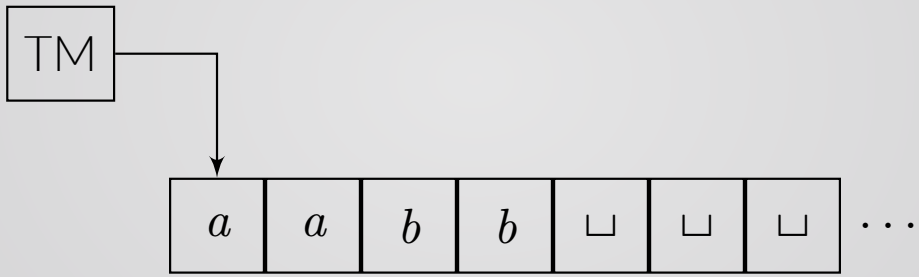
1. **Tape**: An infinite ”tape” of memory cells
2. **Head**: A read/write head pointing at a single memory cell on the tape. The head can move left and right along the tape
3. **Control**: A finite automaton that controls movements of the head, reading from and writing to the tape. It has explicit accept/reject states and the TM stops (halts) whenever it reaches one.

Technically the tape is not infinite in both directions: it has a beginning, but continues forever. Initially the input string appears on the tape starting at the beginning. All cells beyond the input are marked ␣, meaning **empty**.

Western Engineering

Turing Machine

A **Turing Machine** is essentially a finite automaton with the ability to read and write from a memory tape:

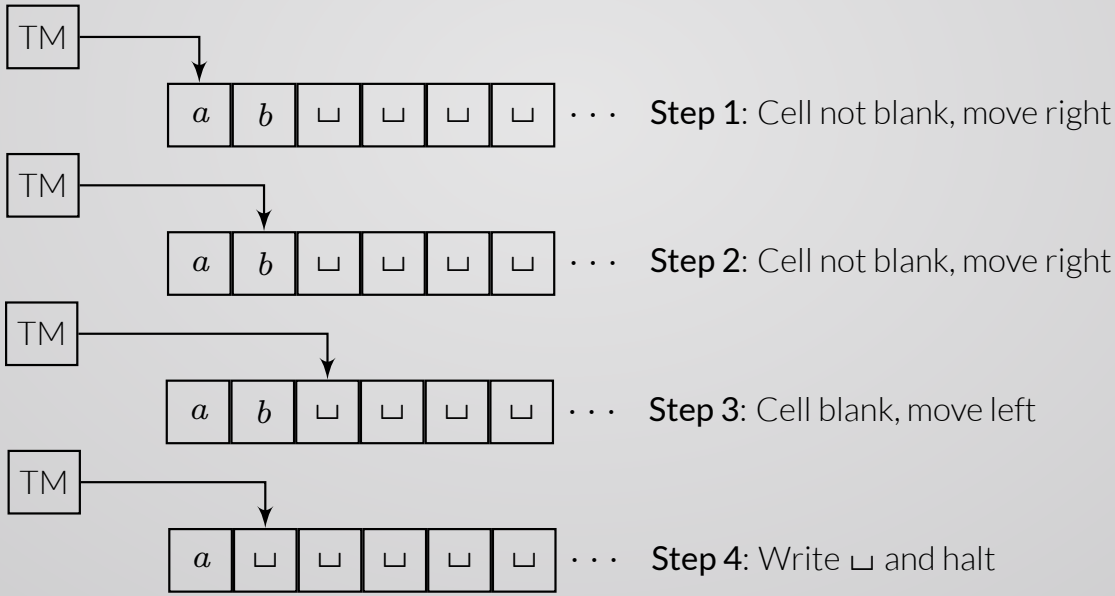


As the Turning machine reads a character from the tape, it may move its head left or right, and may write a new character to the tape.

Western Engineering

Turing Machine Example

Example: A Turing Machine erasing the last symbol on the input tape:



Western Engineering

Turing Machine Definition

In order to be more self-consistent with the defnitions for NFAs and PDAs (and to make your life a little easier) we'll follow along the lines of the Peter Linz definition of TMs instead of Sipser's (see comparison here):

1. The tape may extend infinitely in *either direction*, and the read head is free to move beyond the input in either direction
2. Instead of having a single final *accept* state, you can may allow $F \subseteq Q$ to be a set of accept states.
3. Instead of having a single, explicit *reject* state, you can assume (like in NFAs and PDAs) that input is rejected if there's no transition out of the current state

You may leave it to faith (or to an intrepid Google search) that two definitions are essentially equivalent.

Western Engineering

Turing Machine Definition

Definition 1 (Turing machine)**.**

A **Turing machine** is a 7-tuple:

1. Q: Finite set of states
2. Σ: Input alphabet (not containing blank symbol $\sqcup$)
3. Γ: Taple alphabet where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Sigma \to Q \times \Gamma \times \{L, R\}$: Transition function
5. $q_0 \in Q$: Start state
6. $F \subseteq Q$: Set of final/accept states

Western Engineering

Example 1: String-copy language

Recall the "string-copy" language $L = \{ww : w \in \{0,1\}^*\}$. Recall from the pumping lemma for context-free languages that L is not context-free, meaning there is no CFG that can generate this language, or similarly a PDA that can recognize it. Let's now see how a Turing machine is able to recognize this langauge.

As a simpifying assumption, let's assume there's a delimiter "#" between the two sub strings, i.e.,

$$L = \{w\#w : w \in \{0,1\}^*\}$$

Western Engineering

Example 1: Solution Strategy

1. Move back and forth across the tape to matching positions on either side of the delimier
2. If the symbols don't match, or no delimiter is found, reject (and halt)
3. In order to keep track of which comparisons you've made already, cross out symbols on the tape, e.g., by replacing a symbol with "X."
4. When all the symbols on the right of the '#' have been crossed out, check if there are any uncrossed symbols left on the right. If so, *reject*. Otherwise *accept*.

Western Engineering

Example 1: Computation flow

The notation $\check{a}$ is used to denote the current location of the head.
Let s=011000#011000:

$\check{0}$	1	1	0	0	0	#	0	1	1	0	0	0	⊔	...	begin
X	$\check{1}$	1	0	0	0	#	0	1	1	0	0	0	⊔	...	cross out
X	1	1	0	0	0	#	$\check{X}$	1	1	0	0	0	⊔	...	cross matching position
$\check{X}$	1	1	0	0	0	#	X	1	1	0	0	0	⊔	...	find last position
X	$\check{X}$	1	0	0	0	#	X	1	1	0	0	0	⊔	...	cross out next
						⋮									
X	X	X	X	X	X	#	X	X	X	X	X	X	$\check{\sqcup}$	...	accept

Western Engineering

Example 1: State transition

Let's draw the state transition diagaram for L:

(in class - see Example 3.9 in course text)

Western Engineering

Related Languages

We saw the strategy for recognizing $L = \{w\#w : w \in \{0, 1\}\}$.
What about arbitrary copies? I.e.,

$$L = \{w\#w\#w\#...\}$$

Recall we proved this counting language non context-free:

$$L' = \{a^i b^i c^i : i \geqslant 0\}$$

But with Turing machines, we can follow the strategy of moving back and forth, crossing out symbols at corresponding positions, and like the string copy language, we can design a program to count arbitrary (non-infinite) copies.

Western Engineering

Example 2: Multiplication

Let's see a Turing machine performing mulitplication. Remember, in our model of formal languages, we're not computing the answer k to inputs $i * j$, rather we're inputting ijk and recognizing the language where $i * j = k$. For simplicity, we'll use unary representation. Let argument 1 be i repetitions of a, argument 2 be j repetitions of b and argument 3 be k repetitions of c. The language then becomes:

$$L = \{a^i b^j c^k : ij = k \text{ and } i, j, k \geqslant 1\}$$

(See Example 3.11 in course text)

Western Engineering