St. Francis Xavier University
Department of Computer Science

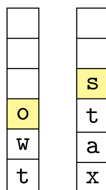
CSCI 356: Theory of Computing
Lecture 3: Beyond Context-Free Languages
Fall 2023

1 Turing Machines

When we introduced pushdown automata, we saw that augmenting our machine with a stack gave it a rudimentary form of memory, and it could use this memory to recognize a larger set of languages. However, despite the fact that the stack has an unbounded capacity (and is therefore able to store as many symbols as it wants), we are still limited by the fact that it's a stack, meaning it can only access the top symbol at any given time.

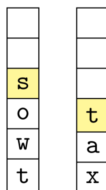
So, how do we overcome this limitation? Let's try adding not one but *two* stacks to our machine! It may not sound like a huge or meaningful change—after all, what can we get with another stack that we didn't already have with the first stack?—but, just like with heads, it turns out that two stacks are better than one.

Suppose we now have two stacks available for us to use. For the purposes of this example, the stacks have been initialized with some symbols already in them.

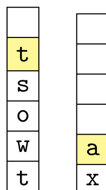


Even with two stacks, we can still only access the top symbol of each stack: in the first stack, we can access the symbol `o`, and in the second stack, we can access the symbol `s`.

But what happens if we use the two stacks in tandem? Suppose we pop one symbol from the second stack, say `s`, and push it to the first stack.



Now, we have access to the symbol `t` that was previously beneath `s` in the second stack, and we haven't lost the symbol `s` since it's safely stored in the first stack! Similarly, if we pop that symbol `t` from the second stack and push it to the first stack, we can get access to the symbol `a` in the second stack:



It's looking like having two stacks is far more meaningful than we might've initially thought. We can push and pop symbols between the two stacks in order to get access to *any* symbol we've stored in the machine's memory, instead of only the most recent symbol at the top.

Indeed, if we took our two stacks and we aligned them horizontally in such a way that the "bottoms" of each stack were on the left and right edges...

t	w	o	s	t	a	x

...we would get a new form of storage: a *tape*.

With a tape, we can move left and right through each cell and access each symbol stored on the tape whenever we want. (This is exactly what we were doing when we pushed and popped symbols between our two stacks.) Additionally, just like our stacks have unbounded capacity, our tape has infinite length. We can write as many symbols to the tape as we want, and we can write them to either the left side or the right side of the tape.[1]

···			i	s	a	t	a	p	e			···

Since we can now access any symbol of the tape that we want, tape cells that do not contain a symbol become an important consideration. With stacks, we need not worry about blank spaces; since we can only push to or pop from the top of a stack, we have no opportunity to leave gaps and so we never encounter a situation where two symbols are separated by a blank space. If we remove a symbol from a tape, however, the symbols to the left and to the right of the removed symbol do not adjust their positions to compensate. Thus, we will notate blank spaces on a tape using the symbol ␣ .

···	␣	␣	i	s	a	t	a	p	e	␣	␣	···

Now that we know the basics of working with tapes, we're ready to replace the stack on our machine with a tape. In doing so, we will obtain one of the most powerful abstract models of computation possible; even more powerful than any real-world computer!

1.1 Definition

The focus of this lecture, the *Turing machine*, is a model of computation that consists of two components: a finite-state control (much like the states of a finite automaton or a pushdown automaton) and an infinite-length tape. The finite-state control keeps track of where we are in the computation, while the tape serves as the machine's memory throughout the computation.

At the beginning of a computation, the tape holds the input word given to the Turing machine, and all other cells of the tape are blank. Since the input word is initially stored on the tape, we can assume that the input alphabet Σ is a subset of the tape alphabet Γ. The input head of the Turing machine starts on the leftmost symbol of the input word. It can move along the tape, and it can both read from and write to cells of the tape. In this way, we can use the tape to store and modify not only the input word, but also any auxiliary information we need to use during the computation.

To model the movement of the Turing machine's input head along the tape, we must account for the direction of movement in the transition function. To figure out the next step of the computation, our transition function will take as input our current state and the tape symbol the input head reads in the current cell, and it will produce as output the state we will transition to, the tape symbol the input head will write to the current cell, and the direction in which the input head will move: one cell leftward (L) or one cell rightward (R).

One other key difference that sets Turing machines apart from finite automata and pushdown automata is in how it accepts or rejects input words. Unlike finite automata or pushdown automata, which can run out of input symbols by reaching the ends of their input words, a Turing machine could theoretically read the

[1] If we take the "two stacks" perspective of thinking about a tape, then writing a symbol to the right side of the tape would require us to move all existing symbols to the first stack and push the new symbol to the second stack. Writing to the left side of the tape is similar.

symbols on its tape as many times as it wants. Therefore, we must fix two special "accept" and "reject" states where, whenever the computation of the Turing machine enters one of those states, it immediately halts the computation and accepts or rejects the input word accordingly. Note that if the Turing machine doesn't visit either of these states during its computation, then it will continue to compute indefinitely.

Apart from these changes, the formal definition of a Turing machine is quite similar to our definitions for finite automata and pushdown automata.

Definition 1 (Turing machine)**.** A Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

- Q is a finite set of *states*;
- Σ is the *input alphabet* (where $\sqcup \notin \Sigma$);
- Γ is the *tape alphabet* (where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$);
- $\delta : (Q \setminus \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*;
- $q_0 \in Q$ is the *initial* or *start state*;
- $q_{\text{accept}} \in Q$ is the *final* or *accepting state*; and
- $q_{\text{reject}} \in Q$ is the *rejecting state.*

Example 2. Consider the language $L_{\text{a=b}} = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$. Even though we know this language is context-free, and is therefore recognized by a pushdown automaton, let's construct a Turing machine recognizing the language.

The idea behind our Turing machine is as follows. Given an input word of the form $\mathtt{a}^n\mathtt{b}^n$ on the tape, the input head will move back and forth, replacing all as with Xs and all bs with Ys. In a sense, the input head is "marking" as and bs as it sees them. Each time the input head replaces an a with an X, it will move rightward in an attempt to find a matching b that it can replace with a Y. The input head will then move leftward and repeat the process until no more as remain.

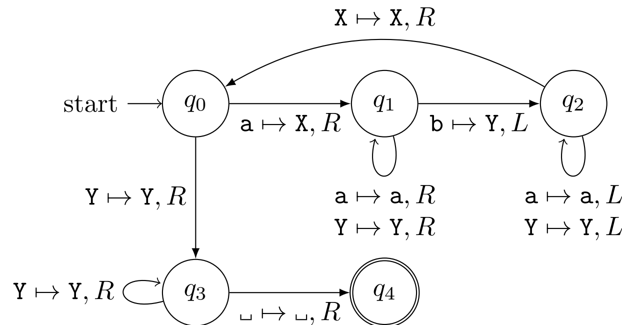
We formally define the Turing machine as follows:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_R\}$;
- $\Sigma = \{\mathtt{a}, \mathtt{b}\}$;
- $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{X}, \mathtt{Y}, \sqcup\}$;
- $q_0 = q_0$;
- $q_{\text{accept}} = q_4$;
- $q_{\text{reject}} = q_R$; and
- δ is specified by the following table:

Γ	a	b	X	Y	$\sqcup$
q_0	$(q_1, \mathtt{X}, R)$	—	—	$(q_3, \mathtt{Y}, R)$	—
q_1	$(q_1, \mathtt{a}, R)$	$(q_2, \mathtt{Y}, L)$	—	$(q_1, \mathtt{Y}, R)$	—
q_2	$(q_2, \mathtt{a}, L)$	—	$(q_0, \mathtt{X}, R)$	$(q_2, \mathtt{Y}, L)$	—
q_3	—	—	—	$(q_3, \mathtt{Y}, R)$	$(q_4, \sqcup, R)$
q_4	×	×	×	×	×
q_R	×	×	×	×	×

Note that we can't have any transitions from either state q_4 or q_R, since those are the accepting and rejecting states, respectively. Instead of writing transitions to state q_R, we just assume that any undefined transition in the table (—) automatically leads to state q_R.

We can draw this Turing machine graphically, just like a finite automaton or a pushdown automaton. To reduce the number of transitions we need to draw, we will omit the state q_R and all transitions leading to it, and we will just assume (again) that all transitions not included automatically lead to state q_R. The Turing machine looks like the following:



Now, let's suppose we give the input word aaabbb to this Turing machine. The input head starts its computation in state q_0 on the leftmost symbol of the input word. Moving from the top to the bottom of each column, we highlight the state of the machine and the input head's current tape cell at each step.

⋯ ␣ **a** a a b b b ␣ ⋯ — q_0

⋯ ␣ X **a** a b b b ␣ ⋯ — q_1

⋯ ␣ X a **a** b b b ␣ ⋯ — q_1

⋯ ␣ X a a **b** b b ␣ ⋯ — q_1

⋯ ␣ X a **a** Y b b ␣ ⋯ — q_2

⋯ ␣ X **a** a Y b b ␣ ⋯ — q_2

⋯ ␣ **X** a a Y b b ␣ ⋯ — q_2

⋯ ␣ X **a** a Y b b ␣ ⋯ — q_0

⋯ ␣ X X **a** Y b b ␣ ⋯ — q_1

⋯ ␣ X X a **Y** b b ␣ ⋯ — q_1

⋯ ␣ X X a Y **b** b ␣ ⋯ — q_1

⋯ ␣ X X a **Y** Y b ␣ ⋯ — q_2

⋯ ␣ X X **a** Y Y b ␣ ⋯ — q_2

⋯ ␣ X **X** a Y Y b ␣ ⋯ — q_2

⋯ ␣ X X **a** Y Y b ␣ ⋯ — q_0

⋯ ␣ X X X **Y** Y b ␣ ⋯ — q_1

⋯ ␣ X X X Y **Y** b ␣ ⋯ — q_1

⋯ ␣ X X X Y Y **b** ␣ ⋯ — q_1

⋯ ␣ X X X Y **Y** Y ␣ ⋯ — q_2

⋯ ␣ X X X **Y** Y Y ␣ ⋯ — q_2

⋯ ␣ X X **X** Y Y Y ␣ ⋯ — q_2

⋯ ␣ X X X **Y** Y Y ␣ ⋯ — q_0

⋯ ␣ X X X Y **Y** Y ␣ ⋯ — q_3

⋯ ␣ X X X Y Y **Y** ␣ ⋯ — q_3

⋯ ␣ X X X Y Y Y **␣** ⋯ — q_3

⋯ ␣ X X X Y Y Y ␣ **⋯** — q_4

Since the computation halts in the accepting state q_4, we know that the machine accepts the word aaabbb.

1.2 Configurations and Accepting Configurations

You may have noticed in the previous example that the computation of the Turing machine on the given input word took up a *lot* of space on the page. This is because we had to draw the current state, the entire tape, and the position of the input head on the tape, all at each step. Fortunately, however, there is a more concise way to present this information.

All we need to specify a particular stage of the computation is the current state, the current tape contents, and the current input head position, and we can represent all of this using a single sequence of symbols. This sequence is called a *configuration* of the Turing machine. If the Turing machine is currently in a state q, its tape contains the symbols $X_1 X_2 \cdots X_{i-1} X_i \cdots X_{n-1} X_n$, and its input head is at cell i of the tape, then the configuration of the Turing machine at this moment is written

$$X_1 X_2 \cdots X_{i-1} q X_i \cdots X_{n-1} X_n.$$

If we can get from a configuration C_i to a configuration C_{i+1} in a single computation step, then we say that C_i *yields* C_{i+1} and we write $C_i \vdash C_{i+1}$. Formally, given $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, and $q_i, q_j \in Q$, we say that uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$. We can define the notion of "yields" for leftward moves similarly.

Example 3. Recalling the Turing machine's computation from the previous example, the first five configurations of the machine are

$$q_0\texttt{aaabbb} \vdash \texttt{X}q_1\texttt{aabbb} \vdash \texttt{Xa}q_1\texttt{abbb} \vdash \texttt{Xaa}q_1\texttt{bbb} \vdash \texttt{Xa}q_2\texttt{aYbb} \vdash \cdots$$

By our earlier definition, we know that a Turing machine includes dedicated accept and reject states, and we know that the machine's computation either accepts or rejects once it enters the appropriate state. With the notion of configurations, we can specify exactly what it means for a Turing machine to accept or reject its input word.

If we have a Turing machine $\mathcal{M}$ and an input word w, then we say that the *start configuration* of $\mathcal{M}$ on w is $q_0 w$. In this configuration, $\mathcal{M}$ is in its initial state q_0, and the input head of $\mathcal{M}$ is on the leftmost symbol of the input word.

Likewise, an *accepting configuration* is one where the current state of $\mathcal{M}$ is q_{accept}, and a *rejecting configuration* is one where the current state of $\mathcal{M}$ is q_{reject}. Note that, in either configuration, we only care about the state and not the tape contents. This is because once we enter the accepting or rejecting state, the computation immediately halts, and so the tape contents don't have any effect on the accepting or rejecting configuration.

We can now formally define what it means for the Turing machine $\mathcal{M}$ to accept its input word w.

Definition 4 (Accepting computation of a Turing machine)**.** Let $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine, and let w be an input word. The Turing machine $\mathcal{M}$ accepts the input word w if there exists a sequence of configurations $C_1, C_2, \ldots, C_n$ satisfying the following conditions:

1. C_1 is the start configuration of $\mathcal{M}$ on w;
2. $C_i \vdash C_{i+1}$ for all $1 \leq i \leq (n-1)$; and
3. C_n is an accepting configuration.

We can write a similar definition for a rejecting computation of a Turing machine by considering rejecting configurations in the third condition.

1.3 Language of a Turing Machine

Just as with other types of automata, every Turing machine recognizes some language. The set of all input words accepted by a Turing machine $\mathcal{M}$ is referred to as the language of the machine $\mathcal{M}$, written $L(\mathcal{M})$. However, the class to which some Turing machine's language belongs is determined by the machine's behaviour on each input word.

We noted earlier on that, unlike with finite automata and pushdown automata, a Turing machine cannot run out of input symbols. Indeed, it can read the symbols on its tape as many times as it wants. The machine's computation immediately halts once the machine enters either the accepting or the rejecting state, but there's no guarantee that it will ever enter either of these states during its computation. We must account for the possibility that the machine simply never ends its computation; that is, the machine enters a *loop*.

Taking this outcome into account, we see that Turing machines can recognize two "types" of languages: languages where the Turing machine always gives us an accept/reject answer for each input word, and languages where the Turing machine might enter a loop and give no answer for certain input words.

Decidable Languages

Let's first focus on the scenario where the machine always either accepts or rejects every input word its given. If the Turing machine accepts all words that belong to its language and rejects all words that don't belong to its language, then we say that the machine *decides* its language.

Definition 5 (Decidable language)**.** Given a Turing machine $\mathcal{M}$, we say that the language $L(\mathcal{M})$ is decidable if,

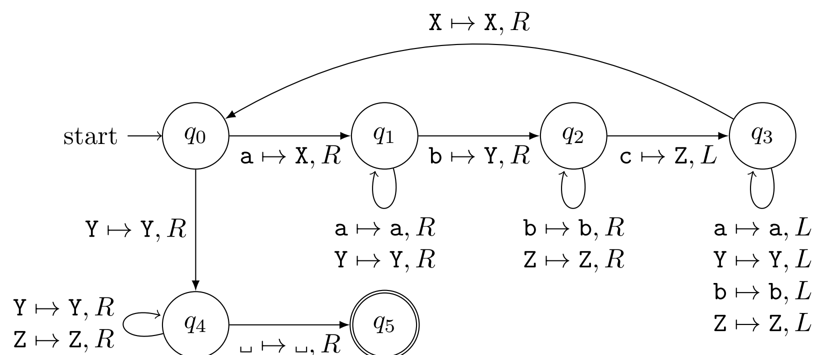
- whenever $w \in L(\mathcal{M})$, then $\mathcal{M}$ accepts w; and
- whenever $w \notin L(\mathcal{M})$, then $\mathcal{M}$ rejects w.

We denote the class of decidable languages by D. In the literature, the class of decidable languages is sometimes called the class of *recursive languages*, where the term "recursive" comes from the origins of computer science and its connections to recursive functions and sets.

Example 6. Consider the language $L_{\text{a=b=c}} = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 1\}$. We know that this language is non-context-free, so we can't construct a pushdown automaton recognizing the language. Let's instead construct a Turing machine recognizing the language.

Our Turing machine will function in much the same way as the machine we constructed to recognize words of the form $\mathtt{a}^n\mathtt{b}^n$; moving from left to right, we will match symbols up to one another by replacing the symbols on the tape.

Suppose our alphabets are $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{X}, \mathtt{Y}, \mathtt{Z}, \sqcup\}$. The Turing machine, then, will look like the following:



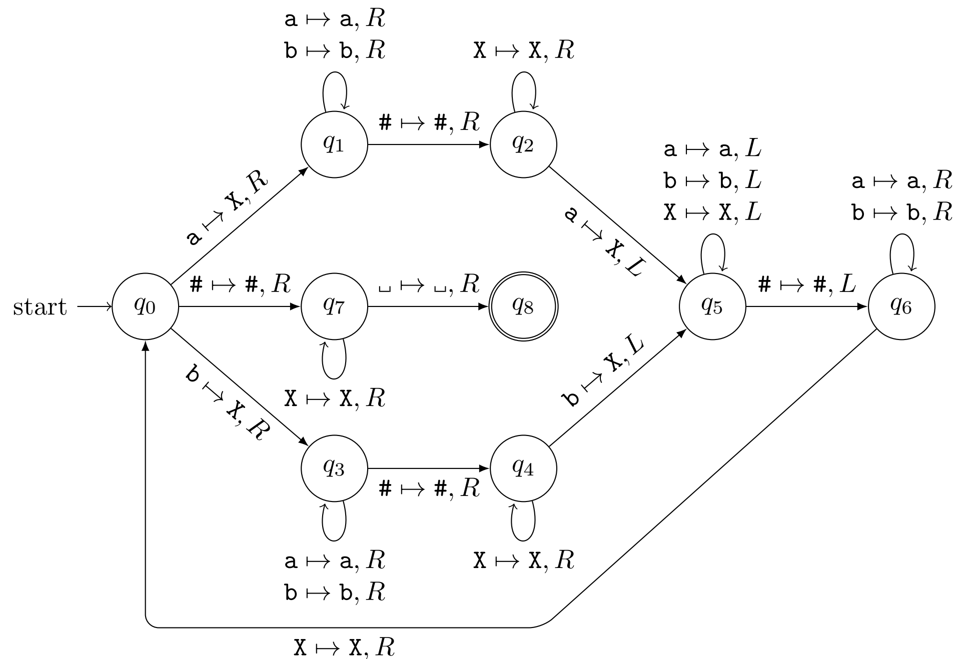
This Turing machine decides the language $L_{\text{a=b=c}}$ because (i) if some input word w belongs to this language, then the machine will accept it; and (ii) if some input word w does not belong to this language, then the machine will (implicitly) go to the rejecting state q_{reject}.

Example 7. Consider the language $L_{\text{double\#}} = \{w\#w \mid w \in \Sigma^*\}$. This language is very similar to the language L_{double}, which we previously showed was non-context-free. The key difference with this language, though, is the presence of a separator symbol # between the two occurrences of w.

Let's construct a Turing machine for the language $L_{\text{double\#}}$. The idea behind this Turing machine is to move back and forth between each copy of w, marking off each symbol in the first copy and using the states of the

machine to "remember" this symbol as we move to the second copy. If the machine is able to match every symbol in both copies of the word, then it accepts. Otherwise, it (implicitly) rejects.

Suppose our alphabets are $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Gamma = \{\mathtt{a}, \mathtt{b}, \mathtt{X}, \sqcup\}$. The Turing machine will look like the following:



This Turing machine decides the language $L_{\text{double}\#}$ because it accepts all words of the form $w\#w$ and (implicitly) rejects all other words.

Semidecidable Languages

Of course, there's no requirement that a Turing machine always either accepts or rejects every input word it's given. As we mentioned before, it's possible that the machine could enter a loop. In this case, then, we can only get a definite answer from the machine if the input word belongs to the machine's language. If a Turing machine accepts all and only those words that belong to its language, then we say that the machine *semidecides* its language.[2]

Definition 8 (Semidecidable language)**.** Given a Turing machine $\mathcal{M}$, we say that the language $L(\mathcal{M})$ is semidecidable if,

- whenever $w \in L(\mathcal{M})$, then $\mathcal{M}$ accepts w; and
- whenever $w \notin L(\mathcal{M})$, then either $\mathcal{M}$ rejects w or $\mathcal{M}$ enters a loop.

We denote the class of semidecidable languages by SD. In the literature, the class of semidecidable languages is sometimes called the class of *recognizable languages* or *recursively enumerable languages.* The term "recursively enumerable" again comes from a connection to mathematics and the notion of recursively enumerable sets. In a machine-oriented context, a recursively enumerable language is one for which a Turing machine can list (or *enumerate*) every word in the language.

We can come up with any number of artificial examples of semidecidable languages, but few such examples are interesting; in fact, it's often the case that making a small change to the Turing machine recognizing such a language results in that language becoming decidable anyway. Therefore, we'll skip the examples for now. In the next lecture, we'll focus on some more natural examples of semidecidable languages, where "natural" means that the language models some inherent property or quality of the machine that recognizes it.

[2]We say the language is "semidecidable" because the Turing machine is only able to decide in the positive case; that is, when the input word belongs to the machine's language.