ECS 120 Lesson 16 – Turing Machines, Pt. 2

Oliver Kreylos

Friday, May 4th, 2001

In the last lesson, we looked at Turing Machines, their differences to finite state machines and pushdown automata, and their formal definition. Today we will concentrate on the computation of Turing Machines. First, let us recollect the informal notion of how a Turing Machine processes an input word:

1. Before a Turing Machine starts computation, the input word $w = w_1 w_2 \ldots w_n \in \Sigma^*$ will be written into the leftmost n cells of the work tape. All other tape cells will be filled with the special blank symbol $\sqcup \in \Gamma$. The read/write head will be located over the leftmost tape cell, and the state control will be in the start state q_0.

2. In each step of computation, the machine will read the character $a \in \Gamma$ located underneath the read/write head and transition from its current state $q_1 \in Q$ guided by the transition function δ: If $\delta(q_1, a) = (q_2, b, D)$, then the machine will transition to state $q_2 \in Q$, will replace the character in the tape cell underneath the read/write head by the tape character $b \in \Gamma$, and will move the read/write head one cell to the left if $D = \mathrm{L}$ or one cell to the right if $D = \mathrm{R}$. If the read/write head is currently located over the leftmost tape cell, and $D = \mathrm{L}$, then the read/write head will remain in its current position.

3. To finish the computation, the machine must enter either the accept state $q_{\text{accept}} \in Q$ or the reject state $q_{\text{reject}} \in Q$. In the former case, the machine accepts the input word; in the latter case, it rejects. If the machine never enters either the accept state or the reject state, computation will continue forever and the machine will never halt.

1 Formal Definition of Turing Machine Computation

To formalize our understanding of how a Turing Machine computes, we have to find a way of describing its current configuration. In the study of PDAs, we encountered *instantaneous descriptions*, which exactly describe the status of computation of a PDA. We will use the same mechanism for Turing Machines as well. To specify a Turing Machine's configuration, we have to know:

- The current state it is in,
- The current contents of the work tape, and
- the current position of the read/write head.

As for PDAs, we will encode the work tape's contents as a string over Γ. Since all the tape cells to the right of the rightmost position the read/write head ever was in are filled with blank symbols, we will not explicitly include those into the tape string. To encode the read/write head's position, we will split the tape string into two parts $u, v \in \Gamma^*$; the first, u, containing all characters from tape cells to the left of the read/write head's current position, and the second, v, containing the characters underneath the tape head and to the right. Together with the machine's current state, an instantaneous description (ID) for a Turing Machine is a triple $(u, q, v) \in \Gamma^* \times Q \times \Gamma^*$, where the current state is q, the current tape content is $uv\sqcup^*$ (u followed by v followed by any number of blank symbols), and the read/write head's current position is over the first character of v. For notational convenience, and to visualize the tape contents and the read/write head's position, an ID for a Turing Machine is often written as "$u\ q\ v$," just concatenating the left tape string, the current state and the right tape string. See Figure 1 for an illustration how a Turing Machine's configuration is represented by an instantaneous description.

We can now define computation as moving from one instantaneous description to the other. We will express that an ID can be reached from another ID in one step by the turnstile relation $\vdash$, similarly to PDAs. We define the turnstile relation by the following cases:

1. $(u, q_1, av) \vdash (ub, q_2, v)$ if and only if $\delta(q_1, a) = (q_2, b, \mathrm{R})$. This is the case where the machine is in state q_1, reads the character a, and the

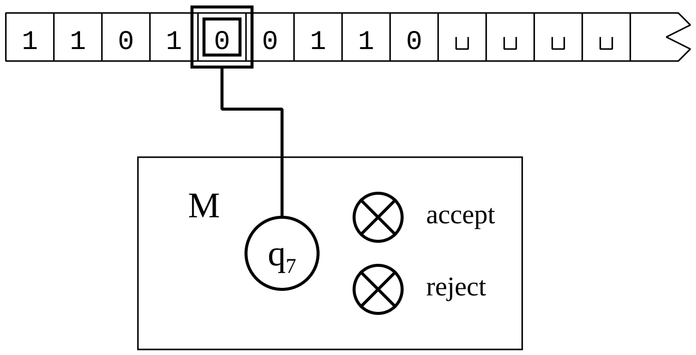

Figure 1: A Turing Machine M with the current configuration `1101` q_7 `00110`.

transition function tells it to transition to state q_2, replace the read character by b, and move the read/write head to the right. In this case, the just written character will be the last one to the left of the read/write head, i. e., it will be the last character of the left tape string.

2. $(uc, q_1, av) \vdash (u, q_2, cbv)$ if and only if $\delta(q_1, a) = (q_2, b, \mathrm{L})$. This is the case where the machine is in state q_1, reads the character a, and the transition function tells it to transition to state q_2, replace the read character by b, and move the read/write head to the left. In this case, the just written character will be the first one to the right of the read/write head, i. e., it will be the second character of the right tape string, and the former last character of the left tape string will be the next character underneath the read/write head.

3. $(\epsilon, q_1, av) \vdash (\epsilon, q_2, bv)$ if and only if $\delta(q_1, a) = (q_2, b, \mathrm{L})$. This is the special case where the read/write head is at the left end of the tape, the machine is in state q_1, reads the character a, and the transition function tells it to transition to state q_2, replace the read character by b, and move the read/write head to the left. In this case, the read/write head will not move but stay in its position; this means, the just written character is going to be the next character underneath the read/write head.

This definition does not handle the case when the right tape string is the empty string ϵ. This means the machine is past the right end of its input; since the tape is not limited to the right, and all tape cells are filled with the blank symbol, we will just append another blank symbol to the end of the right tape string, and continue as usual. In this way, the right tape string will always be as long as needed.

We can now define the extended turnstile relation $\vdash^*$ as usual, as the transitive closure of the turnstile relation $\vdash$:

Base Case $(u, q, v) \vdash^* (u, q, v)$. Any ID can be reached from itself by zero computation steps.

Inductive Case If $(u_1, q_1, v_1) \vdash^* (u_2, q_2, v_2)$, and $(u_2, q_2, v_2) \vdash (u_3, q_3, v_3)$, then $(u_1, q_1, v_1) \vdash^* (u_3, q_3, v_3)$. If ID_2 can be reached from ID_1 in n computation steps, and ID_3 can be reached from ID_2 in exactly one computation step, then ID_3 can be reached from ID_1 in $n + 1$ computation steps.

We also have to define some special instantaneous descriptions:

- (ϵ, q_0, w), corresponding to the machine being in its start state, the input word contained the leftmost tape cells, and the read/write head positioned over the leftmost tape cell, is called the *initial configuration.*
- Any configuration $(u, q_{\text{accept}}, v)$ is called an *accepting configuration.*
- Any configuration $(u, q_{\text{reject}}, v)$ is called a *rejecting configuration.*
- A configuration is called a *halting configuration*, if it is either an accepting or a rejecting configuration.

Using the definitions above, a Turing Machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ *accepts* a word $w \in \Sigma^*$, if and only if there exist $u, v \in \Gamma^* : (\epsilon, q_0, w) \vdash^* (u, q_{\text{accept}}, v)$, in other words, if an accepting configuration can be reached from the initial configuration. Turing Machine M *rejects* a word $w \in \Sigma^*$, if and only if there exist $u, v \in \Gamma^* : (\epsilon, q_0, w) \vdash^* (u, q_{\text{reject}}, v)$.

2 The Language of a Turing Machine M

The language of a Turing Machine M is the set of strings the machine accepts: $L(M) = \left\{ w \in \Sigma^* \;\middle|\; \exists u, v \in \Gamma^* : (\epsilon, q_0, w) \vdash^* (u, q_{\text{accept}}, v) \right\}$. We also say that $L(M)$ is the language that is *recognized by* Turing Machine M. For finite state machines and pushdown automata, there were only two possible outcomes for any computation: The machine would either accept or reject. For Turing

Machines, however, there is a third possible outcome: The machine could continue computing forever, never reaching a halting configuration. This leads to two different classes of languages defined by Turing Machines:

- A language $A \subset \Sigma^*$ is called *Turing-recognizable* or *recursively enumerable* if and only if it can be recognized by some Turing Machine M, i. e., if it is the language $L(M)$. This means machine M accepts every word $w \in A$; but if it is given a word $w \notin A$, it can either reject or never halt.

- A language $A \subset \Sigma^*$ is called *Turing-decidable* or just *decidable* if and only if it is the language of a Turing Machine M that halts on all $w \in \Sigma^*$; in other words, M accepts every word $w \in A$ and rejects every word $w \notin A$. A Turing Machine that halts on all input words is called a *decider*.

By this definition, every Turing-decidable language is automatically a Turing-recognizable language, but the reverse is not true. We will see that there are (important) languages that can be recognized by Turing Machines but cannot be decided.

3 Turing Machines and the Chomsky Hierarchy

So far, we have seen that there are classes of machine models matching the lower two levels of the Chomsky Hierarchy: The languages accepted by finite state machines are exactly the languages in CH–3, and the languages accepted by (nondeterministic) pushdown automata are exactly the languages in CH–2. As it turns out, Turing Machines match the upper two levels of the Chomsky Hierarchy: The class of languages recognized by general Turing Machines, the Turing-recognizable or recursively enumerable languages, are exactly the languages in CH–0.

We can now restrict the Turing Machine model by requiring that certain Turing Machines will never access the tape beyond the right end of the input word given to them. Essentially, these are machines that only have a limited amount of tape – exactly the amount that is needed to write down the input. These Turing Machines, called *linearly limited Turing Machines*, can exactly accept the languages in CH–1.

Furthermore, for every language in CH–1, there is a deciding Turing Machine that determines whether a given word is in the language or not. Therefore, CH–1 is a subset of the class of decidable languages. As we will see later, there are decidable languages which cannot be decided by Turing Machines never writing past the end of their input; therefore, CH–1 is a proper subset of the class of decidable languages. Altogether, the structure of languages that we have seen so far can be illustrated by the inclusion diagram shown in Figure 2.

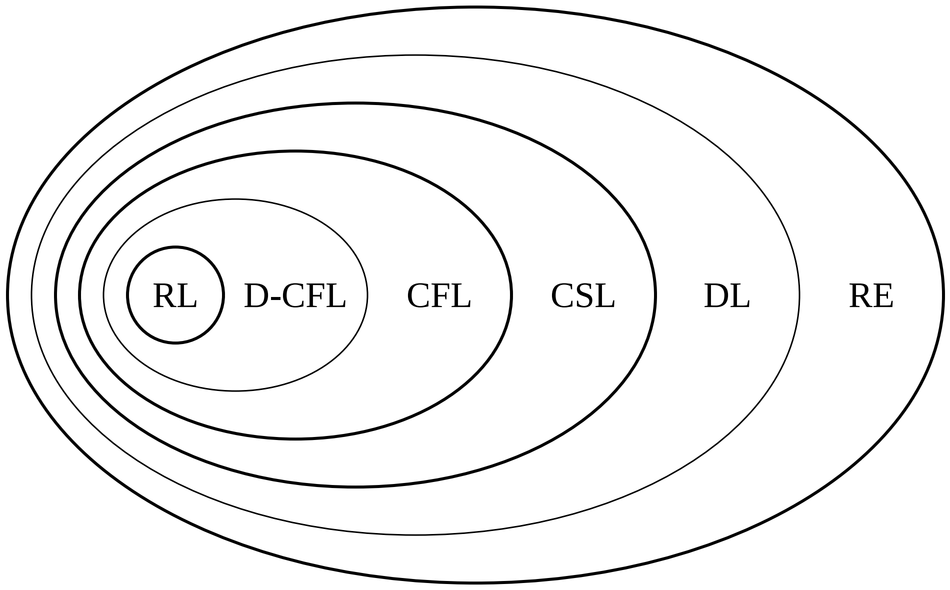

Figure 2: Inclusion diagram of the Chomsky Hierarchy, as we know it at this point. Later, we will discover the fine structure of the class of decidable languages in more detail. Legend: RL: regular languages (CH–3), D–CFL: languages accepted by deterministic PDAs, CFL: context-free languages (CH–2), CSL: context-sensitive languages (CH–1), DL: decidable languages, RE: recognizable languages (CH–0).

4 Nondeterministic Turing Machines

As with finite state machines, Turing Machines also come as a nondeterministic variant. The definition is very similar; the only difference between a nondeterministic Turing Machine and a deterministic one is the definition of the transition function $\delta\colon Q \times \Gamma \to \mathcal{P}\big(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}\big)$. Computation in nondeterministic TMs proceeds as in NFAs: Whenever a current state and read character allow several different transitions in δ, the computation will branch into all those transitions in parallel. The machine is defined to accept a word w, if at least one branch of computation will reach the accept state.

The question now is whether nondeterministic Turing Machines are more powerful than deterministic ones. We have seen before that NFAs are equivalent to DFAs, but that nondeterministic PDAs are indeed more powerful than deterministic PDAs. For Turing Machines, it turns out that nondeterminism does not add to the capabilites of the machine model: For every nondeterministic Turing Machine N, there is an equivalent deterministic Turing Machine D, such that $L(N) = L(D)$.

The proof for this statement constructs a deterministic Turing Machine (DTM) that can simulate all possible branches of computation of a nondeterministic Turing Machine (NTM). This is similar to the way a DFA simulates an NFA, but for Turing Machines, the simulation comes at a price: The computation of the DTM needs much more steps than the computation of the NTM. The reason is that an NTM can perform all branches of computation in parallel, whereas the DTM must simulate the branches serially, one at a time. Since the number of branches of computation can grow exponentially, the number of steps the simulating DTM has to perform can grow prohibitively large for even simple computations. We will explore this phenomenon in detail later in the lecture.

5 Enumerating Turing Machines

The Turing Machines we have seen so far are all *accepting* Turing Machines, i. e., they are fed a word and accept the word if it is an element of the machine's language; otherwise, they either reject or loop. In this way, they are similar to finite state machines or pushdown automata. For the classes of regular and context-free languages, we have also looked at mechanisms *generating* languages: Regular expressions and context-free grammars. Though there exist grammars to generate the languages recognized by Turing Machines, they are never[1] used.

[1]Well, hardly ever…

To generate Turing-recognizable languages, we use a different version of Turing Machine instead. These special machines, called *enumerators*, are like printers: They start with an empty work tape, and write all words of their language onto the work tape, one at a time, in any order, and maybe with repetition. This means, that if a language has an infinite number of words in it, an enumerator printing it can never halt. But since a language can have at most a countable number of words in it, we know that an enumerator for any language will print any word in the language sooner or later (or very much later); but no word in the language will never be printed. We define the language of an enumerator M as $L(M) = \big\{\, w \in \Sigma^* \;\big|\; w \text{ is printed by } M \text{ at some point in time} \,\big\}$.

The languages that can be generated by enumerators are called *Turing-enumerable.* As it turns out, the classes of Turing-recognizable and Turing-enumerable languages are identical. To prove this claim, we show how to convert any enumerator into an acceptor for the same language, and vice versa. The proofs for these two claims will exhibit a structure that is typical for more complicated proofs about Turing Machines. Instead of giving the formal definition of a Turing Machine and proving that it behaves in the intended way, we give a high-level description how to build the intended machine, and argue using that high-level description. Basically, we are building "libraries" of more and more complicated Turing Machines, and argue in terms of their properties, instead of going all the way down to $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$.

5.1 Enumerable Languages $\subset$ Recognizable Languages

Let L be an enumerable language. Then there exists an enumerator E that generates it. To construct a recognizer R for L, we construct a Turing Machine R that will have E as a "subroutine." This new machine will run E, and for every word v that E prints, it will compare v to w and will transition to the accept state if they are identical. If $w \in L$, then it will be printed by E at some point in time (by definition of $L(E)$). At that moment, R will accept. On the other hand, if $w \notin L$, E will never print it; thus, R will never accept and loop. Thus, by definition, L is the language accepted by R.

5.2 Recognizable Languages $\subset$ Enumerable Languages

Let L be a recognizable language. Then there exists a recognizer R that accepts it. To construct an enumerator E for L, we first construct an enumerator E_0 for Σ^*, the set of all words over the alphabet Σ. This enumerator will work by enumerating the words sorted in lexicographical order: Sorted by length of the word, and then by alphabetical order. For example, E_0 for the alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$ would generate Σ^* in the following order: $\{\epsilon, \texttt{0}, \texttt{1}, \texttt{00}, \texttt{01}, \texttt{10}, \texttt{11}, \texttt{000}, \texttt{001}, \ldots\}$. Here is the algorithm for an enumerator for an alphabet $\Sigma = \{s_1, s_2, \ldots, s_n\}$:

1. Print the current word. Initially, the tape is completely blank; therefore, the current word is the empty word.

2. Put the first character of the alphabet, s_1, into the first tape cell and print the current word.

3. Move the read/write head to the rightmost non-blank character.

4. If the current character is s_i, where $1 \leq i < n$, replace it with s_{i+1}, print the resulting word, and repeat from step 3.

5. Otherwise, the current character must be s_n. If the read/write head is not at the left end of the tape, replace the current character with s_1, move the read/write head one to the left, and repeat from step 4.

6. The read/write head is at the left end of the tape. Scan all the way to the first blank symbol, replace it with s_1, print the current word and repeat from step 3.

We will now combine the enumerator E_0 and the recognizer R to form an enumerator E for L. The basic idea is to have E_0 generate all words over Σ in order, and to run R on each of these words in turn. When R accepts a word, the enumerator E will print it. Since every possible word is generated by E_0 at some point in time, every word in L will be fed into R and printed at some point in time. Therefore, E enumerates L.

There is a problem with this approach, though: R is only a recognizer for L, not a decider. This means, that if a word not in L is fed into R, it might never halt and the enumerator would be stuck in a loop, stopping to print words. This problem can be overcome by a strategy called *dovetailing*[2]:

[2]Another bird principle.

Instead of running R to completion on every word printed by E_0, the enumerator E starts another "recognizer process." As long as there are multiple recognizer processes running, every one of them will be executed one computation step at a time in a *round-robin*[3] fashion. Whenever one of those processes finishes in an accept state, the word that was fed into it is printed by E. Since all recognizer processes started on a word $w \in L$ must finish and accept sooner or later, all words in L will be printed – not necessarily in order.

If R were a decider instead of just a recognizer, the construction would have been much easier: Deciders must terminate on all input; therefore, dovetailing is not necessary to make the enumerator work.

[3]Yet another bird principle.