Handout 2: Turing Reductions

Note: You are welcome to borrow language from this document as you write your own Turing reductions. You may even use whole sentences verbatim from this document if you wish.

The purpose of this handout is to provide you with a detailed example of how to formulate, write, and analyze a Turing reduction. We will do this for the first example we covered in class: the reduction from L_{BARBER} to L_{ACC}. First, let's recall the definitions of these two languages. In our notation, M is a Turing machine and $\langle M \rangle$ is the "encoding"—i.e., representation, or description—of M as a string. Recall that

$$L_{\text{BARBER}} = \{\langle M \rangle : M \text{ does not accept } \langle M \rangle\} .$$

That is, the string $\langle M \rangle$ is a member of L_{BARBER}, written $\langle M \rangle \in L_{\text{BARBER}}$, if M, when run its *own description* $\langle M \rangle$ as input, does *not accept* (i.e., it either rejects or loops). Also recall that

$$L_{\text{ACC}} = \{(\langle M \rangle, x) : M \text{ accepts } x\} .$$

That is, $(\langle M \rangle, x)$ is a member of L_{ACC}, written $(\langle M \rangle, x) \in L_{\text{ACC}}$, if M, when run on x as input, *accepts* (i.e., it neither rejects nor loops).

Our goal is to prove that L_{ACC} is undecidable, using the fact (established in class) that L_{BARBER} is undecidable. We do this by giving a Turing reduction from L_{BARBER} to L_{ACC}, i.e., showing that $L_{\text{BARBER}} \leq_T L_{\text{ACC}}$.

Why $L_{\text{BARBER}} \leq_T L_{\text{ACC}}$ proves the undecidability of L_{ACC}.

1. Our goal is to show that since L_{BARBER} is undecidable, L_{ACC} is undecidable as well. To do this, we will instead prove the *contrapositive*: that if L_{ACC} is decidable, then L_{BARBER} is decidable as well.

   [*Optional commentary:* Proving the contrapositive statement seems much more promising, because "undecidable" means the *non-existence* of a decider, and it's often very tricky to prove that a certain object does not exist! By contrast, "decidable" means the *existence* of a decider for the desired language, and to prove existence it is enough to just *construct* such a decider—in this case, using the hypothesis that L_{ACC} is decidable.]

2. To that end, we begin by assuming that L_{ACC} is decidable, which means that there is a Turing machine M_{ACC} that decides L_{ACC}.

3. Next, we give a Turing reduction from L_{BARBER} to L_{ACC}. That is, we *construct* a machine M_{BARBER} that uses M_{ACC} as a subroutine, and then do some *analysis* to prove that M_{BARBER} does indeed decide L_{BARBER}.

   This is what the notation $L_{\text{BARBER}} \leq_T L_{\text{ACC}}$ ("L_{BARBER} Turing-reduces to L_{ACC}") means. More precisely, it means that there is Turing machine that decides L_{BARBER} when given access to a "black-box" or "oracle" that decides L_{ACC}. (See below for elaboration on the concept of an "oracle.")

4. The existence of M_{BARBER} from the previous step means that L_{BARBER} is decidable, which is what we set out to prove.

This four-step template is very common in undecidability proofs, but notice that everything surrounding Step 3 is essentially "boilerplate" that doesn't depend on the specific languages in question. To avoid such repetition in our proofs, we can use the following theorem (proved in class) that abstracts away these steps.

Theorem 0.1. *If $A \leq_T B$ and A is undecidable, then B is undecidable as well.*

It is enough, then, just to execute Step 3, i.e., prove that $L_{\text{BARBER}} \leq_T L_{\text{ACC}}$. Then we simply invoke Theorem 0.1 with $A = L_{\text{BARBER}}$, $B = L_{\text{ACC}}$, and the fact that L_{BARBER} is undecidable, to conclude that L_{ACC} is undecidable, as desired.

It is important to be careful: **we must do the reduction in the right "direction"**. One of the biggest pitfalls in doing reductions is going in the wrong direction! If we instead gave a reduction from L_{ACC} to L_{BARBER}—i.e., proved that $L_{\text{ACC}} \leq_T L_{\text{BARBER}}$—this would *not* imply that L_{ACC} is undecidable.[1]

Constructing a Turing reduction from L_{BARBER} to L_{ACC}.

Our goal is to give a Turing reduction from L_{BARBER} to L_{ACC}. That is, we wish to construct a machine M_{BARBER} that decides L_{BARBER}, using the help of an *oracle* M_{ACC} that decides L_{ACC}. Here is what it means for M_{ACC} to be an oracle that decides L_{ACC}: whenever M_{ACC} is invoked, or "queried," on a string, it correctly identifies whether that string is in L_{ACC}: it accepts if so, and rejects if not. Helpfully, M_{ACC} cannot loop!

Our reduction will be given an input string $\langle M \rangle$ representing a Turing machine, and its goal is to determine whether $\langle M \rangle \in L_{\text{BARBER}}$, i.e., whether $M(\langle M \rangle)$ *does not accept*. To do this, the reduction can query M_{ACC} on any string of interest, and M_{ACC} will correctly answer whether that string is in L_{ACC}. (The reduction can even query M_{ACC} many times on various strings.) More specifically, the reduction can invoke M_{ACC} on *any* string of the form $(\langle M' \rangle, x)$, where M' and x are respectively any Turing machine and string of interest, and M_{ACC} will correctly answer whether M' accepts when run on input x. Our reduction will exploit this "special power" in order to achieve its goal, which is to determine whether its input $\langle M \rangle$ is in L_{BARBER}.

The *key insight* is that the reduction can ask M_{ACC} whether M accepts when run on its own description $\langle M \rangle$, i.e., whether $(\langle M \rangle, \langle M \rangle) \in L_{\text{ACC}}$. This is exactly what the reduction needs to know in order to determine whether $\langle M \rangle \in L_{\text{BARBER}}$. Here is the precise construction of the reduction M_{BARBER}.

$M_{\text{BARBER}}(\langle M \rangle)$:
  Query $M_{\text{ACC}}(\langle M \rangle, \langle M \rangle)$ and output the opposite answer:
    if it accepts **then** reject;
    if it rejects **then** accept.

Proving the correctness of (i.e., analyzing) the reduction.

To prove that the reduction is correct, we need to show that our machine M_{BARBER} does indeed decide the language L_{BARBER}. By definition of "decides," we need to show both of the following:

[1] It would imply that if L_{BARBER} is decidable, then so is L_{ACC}. But we already have established that L_{BARBER} is *not* decidable, so the premise is false, and therefore the statement is vacuous.

1. If $\langle M \rangle \in L_{\text{BARBER}}$, i.e., if $M(\langle M \rangle)$ *does not accept*, then $M_{\text{BARBER}}(\langle M \rangle)$ *accepts*.
2. If $\langle M \rangle \notin L_{\text{BARBER}}$, i.e., if $M(\langle M \rangle)$ *accepts*, then $M_{\text{BARBER}}(\langle M \rangle)$ *rejects*.

Proof of 1: Suppose that $M(\langle M \rangle)$ does not accept. Let's trace the execution of $M_{\text{BARBER}}(\langle M \rangle)$: first, it queries $M_{\text{ACC}}(\langle M \rangle, \langle M \rangle)$. By definition of M_{ACC}, because $M(\langle M \rangle)$ does not accept, $M_{\text{ACC}}(\langle M \rangle, \langle M \rangle)$ *rejects*.[2] Finally, M_{BARBER} outputs the opposite answer, so it accepts, as needed.

Proof of 2: Suppose that $M(\langle M \rangle)$ accepts. Then in the execution of $M_{\text{BARBER}}(\langle M \rangle)$, the query to $M_{\text{ACC}}(\langle M \rangle, \langle M \rangle)$ accepts, by the definition of M_{ACC}. Finally, M_{BARBER} outputs the opposite answer, so it rejects, as needed.

An alternative analysis. Another equally valid (and somewhat less repetitive) analysis uses an equivalent definition of "decides," which requires both of the following:

1. M_{BARBER} is a decider, i.e., it halts (does not loop) on every input.
2. $\langle M \rangle \in L_{\text{BARBER}} \iff M_{\text{BARBER}}(\langle M \rangle)$ accepts.

For property 1, observe that M_{BARBER} does not loop (no matter the input), because it simply queries M_{ACC} (which does not loop) and returns the opposite answer.

For property 2, we have that

$$\begin{aligned}
\langle M \rangle \in L_{\text{BARBER}} &\iff M(\langle M \rangle) \text{ does not accept} && \text{(by definition of } L_{\text{BARBER}}) \\
&\iff (\langle M \rangle, \langle M \rangle) \notin L_{\text{ACC}} && \text{(by definition of } L_{\text{ACC}}) \\
&\iff M_{\text{ACC}}(\langle M \rangle, \langle M \rangle) \text{ rejects} && \text{(because } M_{\text{ACC}} \text{ decides } L_{\text{ACC}}) \\
&\iff M_{\text{BARBER}}(\langle M \rangle) \text{ accepts} && \text{(by the code of } M_{\text{BARBER}}).
\end{aligned}$$

When writing a proof in this style, it is very important to ensure that each step holds *if and only if* the previous one does, i.e., the implications must hold in both directions. This is the case for our proof, because all the definitions we have used are "if and only if" conditions.

[2] Very importantly, even though $M(\langle M \rangle)$ might loop, our reduction never actually *runs* $M(\langle M \rangle)$ internally. In addition, M_{ACC} *does not loop* when it is queried, even though it is being "asked about" a computation that does loop! This is the key reason why our reduction does not loop, even if $M(\langle M \rangle)$ does.